Universidad de Alcalá
Escuela Politécnica Superior

Máster Universitario en Ingeniería de Telecomunicación

Trabajo Fin de Máster

Desarrollo de una solución de encaminamiento para tráfico de control in-band en entornos SDN

Autor: Boby Nicusor Constantin

Tutora: Elisa Rojas Sánchez

2020

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Trabajo Fin de Máster

Desarrollo de una solución de encaminamiento para tráfico de control in-band en entornos SDN

Autor: Boby Nicusor Constantin

Tutora: Elisa Rojas Sánchez

Tribunal:

Presidente: D. Isaías Yelmo Martínez

Vocal 1º: D. Lucas Cuadra Rodríguez

Vocal 2º: Dña. Elisa Rojas Sánchez

Agradecimientos

Sólo cerrando las puertas detrás de uno se abren
ventanas hacia el porvenir.

Françoise Sagan

El presente proyecto es el resultado de muchas horas de trabajo y dedicación por parte de todas las personas que se han implicado durante su desarrollo.

En primer lugar quería agradecer a mi tutora Elisa Rojas Sánchez toda la paciencia que ha demostrado tener conmigo durante el desarrollo del proyecto, por aguantar mis entregas a destiempo pero, sobre todo, por darme útiles consejos y directrices siempre que lo he necesitado.

También quiero agradecerle a Isaías Yelmo Martínez todos los consejos y ánimos que me ha dado durante los momentos difíciles que he atravesado desarrollando este proyecto. Sin su ayuda difícilmente habría podido alcanzar los objetivos que se habían marcado al principio del TFM.

A todos los miembros del grupo GIST-Netserv y compañeros de laboratorio por acogerme tan bien y facilitarme enormemente la integración en el grupo. Gracias por las charlas durante los desayunos, las risas en el laboratorio y, sobre todo, gracias por haberme enseñado multitud de cosas nuevas y por hacer que los dos años que hemos pasado juntos hayan sido una gran experiencia en mi vida. Ha sido un placer haber pasado todo este tiempo con vosotros.

Por último, y más importante, agradecer a mi familia y amigos el apoyo que me ha brindado durante esta etapa y la paciencia que han tenido conmigo en los momentos de mayor estrés. Sin ellos no habría podido alcanzar el final del camino. A todas esas personas con las que he compartido grandes momentos durante este desafío, que me han aconsejado, motivado o presionado para que me centrara y terminara el proyecto. A todos vosotros, gracias.

Resumen

En este Trabajo de Fin de Máster se presenta una solución que implementa la lógica necesaria para que el *software-switch* BOFUSS sea capaz de encaminar el tráfico de control de tipo *in-band* haciendo uso de los caminos posibles hasta el controlador generados mediante el protocolo Amaru.

Para poder llevar a cabo el proyecto, se han analizado las tecnologías que abarcan los objetivos del trabajo y así poder determinar las herramientas necesarias para su desarrollo. Una vez se han seleccionado las herramientas y dispositivos necesarios, se ha profundizado en el estudio de estos elementos para lograr comprender su funcionamiento. Este análisis más profundo es el que ha permitido implementar en el switch las distintas modificaciones, con el propósito de lograr que sea capaz de desempeñar las funcionalidades previstas para alcanzar los objetivos del proyecto. Por último, las implementaciones se han testeado en diversos escenarios para comprobar su correcto funcionamiento.

Palabras clave: SDN, switch BOFUSS, pipeline, datapath, OpenFlow, Amaru, in-band, out-of-band, tabla de flujos, match.

Abstract

In this Master Thesis, we review a solution that implements the logic needed for the BOFUSS software-switch to be capable of routing in-band traffic control, by leveraging the possible paths up to the controller generated by the Amaru protocol.

We have reviewed the appropriate technologies in order to determine the necesary tools required to achieve the objectives of the project. Once the necessary tools and devices have been selected, these elements have been studied in depth in order to improve our understanding of them. This deeper analysis has allowed different modifications in the switch, in order to make it capable of running the functionalities planned to achieve the objectives. Finally, the implementation of the project have been tested in various scenarios to check their functional operation.

Keywords: SDN, BOFUSS switch, pipeline, datapath, OpenFlow, Amaru, in-band, out-of-band, flows tables, match.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

Lista de acrónimos

Basic OpenFlow User-space Software Switch	BOFUSS.
Controller-Application Plane Interface	CAPI.
Controller-Data Plane Interface	CDPI.
Media Access Control	MAC.
Open Network Operating System	ONOS.
Open Networking Foundation	ONF.
Quality of Service	QoS.
Secure Socket Layer	SSL.
Software-Defined Networking	SDN.
Virtual Local Area Networks	VLANs.
eXtensible Markup Language	XML.
Tecnologías de la Información y la Comunicación	TIC.

Capítulo 1

Introducción

La mejor manera de predecir el futuro es crearlo.

Abraham Lincoln

1.1 Motivación

Las Redes Definidas por *Software*, en inglés SDN (*Software-Defined Networking*), ofrecen a las organizaciones la posibilidad de agilizar la implementación y distribución de aplicaciones reduciendo los costes en TIC (Tecnologías de la Información y la Comunicación) mediante la automatización del flujo de trabajo. La tecnología SDN permite crear arquitecturas en la nube donde las aplicaciones se distribuyen y gestionan de manera automatizada. Este método de funcionamiento ha permitido incrementar los beneficios de la virtualización en los centros de datos ya que proporciona mayor flexibilidad y utilización de los recursos y además, disminuye los costes de la infraestructura sin aumentar el coste de operación. Para lograr estos objetivos, las Redes Definidas por *Software* delegan la administración de los servicios de red y aplicaciones en plataformas centralizadas y escalables que permiten automatizar el aprovisionamiento y la configuración de toda la infraestructura. Esto hace que las redes SDN ofrezcan una alta velocidad y mayor agilidad a la hora de implementar nuevos servicios y aplicaciones [12].

La adopción de la tecnología SDN se ha acelerado en los últimos años, pasando de representar un volumen de negocio de 10 millones de dólares en 2007 a los 252 millones en el año 2012 [13]. La consultora IDC estimaba en 2016 que el volumen de negocio en torno a este tecnología rondaría los 12.500 millones de dólares en 2020 y que la tasa anual de crecimiento de esta tecnología sería de un 54 % [14].

Las Redes Definidas por *Software* suponen un enfoque que optimiza y simplifica las operaciones de red al vincular más estrechamente la interacción (es decir, el aprovisionamiento, la mensajería y las alarmas) entre las aplicaciones y los servicios y dispositivos de red, ya sean reales o virtualizados. Generalmente se logra empleando un punto de control de red lógicamente centralizado, que a menudo se realiza como un controlador SDN, que se encarga de orquestar, mediar y facilitar la comunicación entre las aplicaciones que desean interactuar con los elementos de red y los elementos de red que desean transmitir información a esas aplicaciones.

Para establecer la comunicación entre la unidad de control y los recursos alojados en el plano de control, actualmente, se han extendido dos modelos de comunicación: *in-band* y *out-of-band*. Como se muestra en la Figura 1.1, el control *out-of-band* se lleva a cabo desplegando una red adicional dedicada exclusivamente a los mensajes de control, lo que requiere cableado, infraestructura e interfaces extra.

el control *in-band*, a diferencia del modo *out-of-band*, utiliza el mismo plano de datos para transmitir los mensajes de la comunicación establecida entre el controlador y el switch, lo que supone un ahorro importante de recursos. Para la implementación de este modo de control los switches deben ser capaces de aplicar métodos de conmutación de paquetes y aprendizaje en capa 2 (capa de enlace) para poder encaminar el trafico *OpenFlow* de los distintos dispositivos [1].

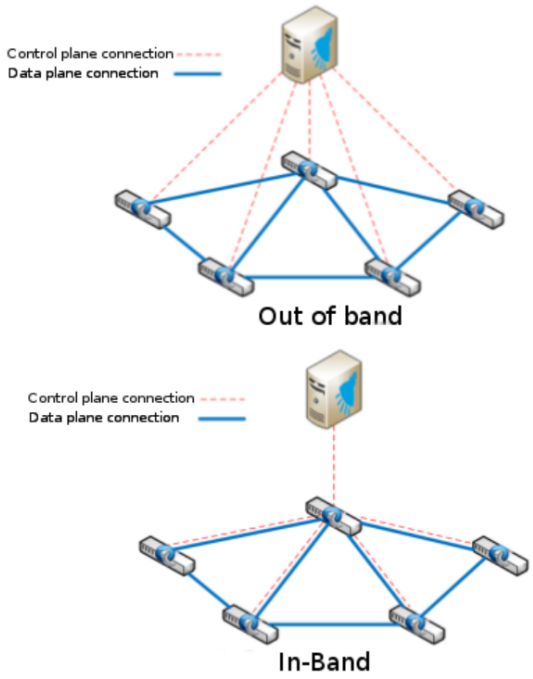


Figura 1.1: Control out-of-band vs control in-band [1]

1.2 Objetivos

El objetivo principal de este proyecto es implementar en un *software switch* un protocolo que permita encaminar tráfico de control *in-band* en entornos SDN. El modo de control *in-band* en entornos SDN, tal y como se ha mencionado en el apartado anterior, implica que el intercambio de mensajes de control entre el switch y el controlador se realice a través de la red que conforma el plano de datos. Este método de intercambio tiene varias implicaciones de seguridad, ya que la comunicación de control se vuelve más accesible al compartir la misma red con los mensajes de datos. Pero, por otro lado, puede permitir la construcción de múltiples rutas de control para dar más robustez al sistema en caso de que alguna de ellas falle o sea atacada. Como ejemplo de protocolo de encaminamiento cabe citar Amaru [8], que es un protocolo que implementa control *in-band* en entornos SDN, proporcionando baja complejidad y alta escalabilidad mediante la implementación de un mecanismo de exploración que busca encontrar todos los caminos posibles entre el controlador o controladores y cualquier nodo del sistema. Esto permitirá que los distintos dispositivos del entorno SDN reconfiguren automáticamente los caminos que le permiten comunicarse con el controlador en caso de que se produzca la caída de alguno de ellos. Por lo que la primera opción será la implementación de este protocolo en un *software switch* como el BOFUSS (*Basic OpenFlow User-space Software Switch*) [15] o el OvS [16].

Los objetivos parciales de este Trabajo de Fin de Máster son los siguientes:

- Estudiar y familiarizarse con la tecnología SDN.

- Estudiar el funcionamiento de los *software-switch* BOFUSS y OvS.
- Estudiar el protocolo Amaru.
- Implementar el control *in-band* mediante el protocolo Amaru en uno de los dos *software-switch*.

1.3 Estructura y contenidos de la memoria

La memoria se ha dividido en cinco capítulos:

- Estado del arte: Este capítulo describe las tecnologías y protocolos más importantes que sustentan este proyecto, así como las herramientas utilizadas más destacadas.
- Análisis y Diseño: Se describe el trabajo que se va a desarrollar y se aportan los motivos que han hecho que se tome esa decisión.
- Desarrollo: Se describen las modificaciones realizadas para alcanzar los objetivos del proyecto.
- Pruebas y resultados: Se detallan las pruebas de funcionamiento que se llevan a cabo y se analizan los resultados obtenidos.
- Conclusiones y líneas futuras: Se abordan las conclusiones obtenidas tras el desarrollo del proyecto y se plantean algunas líneas de mejora de cara al futuro.

Capítulo 2

Estado del arte

Cuéntamelo y me olvidaré. Enséñamelo y lo recordaré.
Involúcrame y lo aprenderé.

Benjamin Franklin

En este capítulo se exponen las principales tecnologías y herramientas ligadas al proyecto, con el objetivo de asentar las bases sobre las que se sustenta el desarrollo que se pretende llevar a cabo mediante la realización del proyecto y acotar el marco tecnológico que abarca.

2.1 Software Defined Networking

Las Redes Definidas por *Software*, o *Software-Defined Networking* (SDN), establecen la separación entre las funciones de control y las funciones de encaminamiento, permitiendo mayor automatización y programabilidad en la red [17].La necesidad de realizar un control más flexible sobre los flujos de tráfico ha sido impulsada por diferentes requisitos que las redes actuales necesitan satisfacer. Uno de los requisitos clave ha sido el incremento de la demanda de servicios de virtualización de servidores. En esencia, la virtualización permite enmascarar los recursos del servidor, incluyendo el numero y la indentidad de cada servidor físico, procesadores y sistemas operativos, para que sean ajenos al usuarios. El enmascarado de los recursos permite particionar una única máquina en múltiples servidores independientes, de tal modo que una única máquina aloje varios servidores virtuales. Además, también posibilita migrar dinámicamente una máquina a otra, por ejemplo, para ofrecer balanceo de carga o para recuperarse de un fallo en el equipo.

La virtualización de servidores se ha convertido en un elemento central para las aplicaciones orientadas al *big data* y en la implementación de infraestructuras para la computación en red. No obstante, este enfoque presenta diferentes problemas debidos a la arquitectura de las redes tradicionales. Uno de los problemas es la configuración de las VLANs (*Virtual Local Area Networks*). Los administradores de red deben asegurarse de que la VLAN utilizada por la máquina virtual se ha asignado al mismo puerto del switch que tiene asignado el servidor físico que está corriendo la máquina virtual. La complicación surge con la migración dinámica de las máquinas virtuales de un servidor a otro, que supone reconfigurar la VLAN cada vez que se traslada a otro equipo. En términos generales, para alcanzar la flexibilidad que requiere la virtualización de los servidores, el administrador de la red debe ser capaz de añadir, eliminar y cambiar los recursos de la red de manera dinámica. Este proceso es complicado de llevar a cabo con

switches convencionales ya que en ellos la lógica de control está ubicada en el mismo lugar que la lógica de conmutación de paquetes.

Otro de los efectos producidos por la virtualización de los servidores es que los flujos de tráfico son substancialmente diferentes respecto al modelo tradicional de cliente-servidor. Normalmente, existe un volumen de tráfico considerable entre servidores virtuales para mantener una imagen consistente de las bases de datos e invocar funciones de seguridad como podría ser el control de acceso. Estos flujos entre servidores varían tanto en localización como en intensidad a lo largo del tiempo, lo que requiere un enfoque flexible a la hora de administrar los recursos de red.

Por otro lado, el incremento del uso por parte de los empleados, de dispositivos móviles, tales como los *smartphones* o *tablets*, para acceder a los recursos de la empresa ha hecho que sea necesario dar una respuesta rápida a la hora de asignar los recursos de red. Los administradores de red deben ser capaces de redimensionar los recursos rápidamente, cambiar la calidad del servicio, QoS (*Quality of Service*), y ajustar los requisitos de seguridad.

Las infraestructuras de red actuales son capaces de responder a cambios en los requisitos para la gestión de los flujos de tráfico, proporcionar niveles diferenciados de calidad de servicio y ofrecer diferentes niveles de seguridad para flujos individuales. Sin embargo, si la red empresarial es grande y/o cuenta con dispositivos de red de diferentes fabricantes, el proceso para rejustar los diferentes parámetros mencionados anteriormente puede alargarse excesivamente en el tiempo debido a que el administrador de red tiene que configurar cada dispositivo por separado y ajustar los parámetros de seguridad y de rendimiento por sesión o por aplicación.

La arquitectura SDN, junto con el estándar *OpenFlow*, proporciona una arquitectura abierta en la cual las funciones de control se separan de los dispositivos de red y se alojan en servidores de control de fácil accesibilidad. Este tipo de configuración permite que la infraestructura subyacente sea transparente para las aplicaciones y los servicios de red, haciendo posible que la red sea tratada como una entidad lógica [18].

2.1.1 Arquitectura SDN

La arquitectura SDN ha ido adquiriendo complejidad a lo largo de su evolución. Para facilitar su comprensión, en la Figura 2.1 se muestra un enfoque a alto nivel de la arquitectura SDN. En ella se observa que la arquitectura está compuesta por diferentes estratos, pudiendo diferenciar claramente tres planos o capas: `plano de aplicación`, `plano de control` y `plano de datos`. Además, las capas contiguas se comunican entre sí mediante interfaces: las interfaces *NorthBound*, también conocidas como CAPI (*Controller-Application Plane Interface*), para la comunicación entre el plano de aplicación y el plano de control, y las interfaces *SouthBound*, también conocidas como CDPI (*Controller-Data Plane Interface*), para la comunicación entre el plano de control y el plano de datos.

Después obtener una enfoque simplificado de la arquitectura SDN, en el siguiente apartado se describen los componentes de una arquitectura que es más fiel a la realidad, y que se muestra en la Figura 2.2.

2.1.1.1 Componentes SDN

A continuación, se describen los principales elementos de la arquitectura SDN:

- **Plano de aplicación**: Este plano aloja las diferentes aplicaciones de red que comunican al controlador de manera explícita sus requisitos de red y el comportamiento de red deseado, a través de las interfaces NBI. Estas aplicaciones tienen una visión abstraída de la red para facilitar la toma de

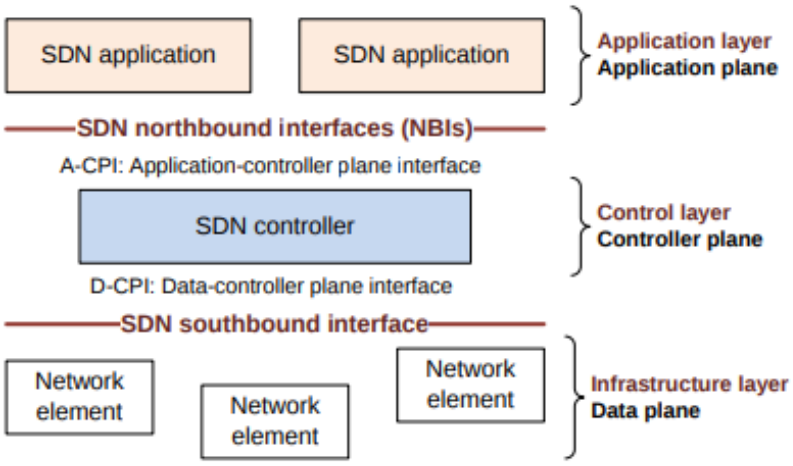


Figura 2.1: Componentes SDN básicos [2]

decisiones en cuanto a sus requisitos. Una aplicación SDN está formada por la lógica de la aplicación y uno o más drivers NBI.

- **Plano de control**: El plano de control está formado por una o más unidades de control o controladores SDN. El controlador es una entidad centralizada lógicamente que se encarga de, en primer lugar, traducir los requisitos del plano de aplicaciones de tal modo que los recursos, que administra del plano de control, puedan comprenderlos y, en segundo lugar, proporcionar a las aplicaciones SDN una visión abstracta de la red. Un controlador SDN está formado por uno o varios agentes CAPI o interfaz `NorthBound`, la lógica de control SDN, y el driver de la interfaz CDPI o interfaz `SouthBound`. La definición del controlador como una entidad lógicamente centralizada no impide que puedan coexistir múltiples controladores, interfaces de comunicación entre ellos e incluso una jerarquía.

- **Plano de datos**: El plano de datos contiene los elementos de red que se encargan de reenviar y procesar el tráfico de los clientes y, además, los recursos necesarios para asegurar una virtualización, conectividad, seguridad, disponibilidad y calidad correctas. Los elementos de red implementan las decisiones tomadas en el plano de control. En principio, dichos elementos no toman decisiones de manera autónoma a menos que la unidad de control lo haya especificado. Los recursos pueden agruparse en entidades o dispositivos lógicos, como puede ser el Datapath SDN.

- **Datapath SDN**: Es un dispositivo de red lógico, que proporciona visibilidad y control sobre sus capacidades de envío y procesamiento de datos. La representación lógica puede abarcar la totalidad o un subconjunto de los recursos de la capa física. Un Datapath SDN está formado por un agente CDPI y un conjunto formado por uno o más motores de reenvío de tráfico y cero o más funciones de procesamiento de tráfico. Uno o más datapaths SDN pueden estar alojados en un solo elemento de red (físico), descrito como una combinación física integrada de recursos de comunicaciones, gestionada como una unidad. Un Datapath SDN también puede definirse a través de múltiples elementos físicos de la red. Esta definición lógica no excluye detalles de implementación tales como el mapeo lógico a físico, la gestión de recursos físicos compartidos, la virtualización o la división en capas (*slicing*) del Datapath SDN, la interoperabilidad con redes no SDN, ni la funcionalidad de procesamiento de datos, que puede incluir funciones de capa 4-7.

- **Interfaces SBI o CDPI**: son las interfaces definidas entre un controlador SDN y un Datapath SDN, que permiten, al menos, controlar las operaciones de reenvío, anunciar las funcionalidades,

obtener estadísticas, y notificar eventos. Un valor de SDN reside en la expectativa de que el CDPI se implemente de forma abierta, neutral para el vendedor e interoperable.

- **Interfaces NBI o CAPI**: son las interfaces definidas entre las aplicaciones y el controlador SDN. Proporcionan una visión abstracta de la red a las aplicaciones y les permite comunicar sus requisitos y observar el comportamiento de la red. Al igual que las interfaces CDPI, también se espera que estas sean implementadas de forma abierta, interoperable y neutral para el vendedor.

- **Agentes y Drivers de interfaz**: cada interfaz se implementa mediante un par driver-agente, el agente representa el lado "sur", inferior o de la infraestructura y el *driver* representa el lado "norte", superior o de la aplicación.

- **Gestión y Administración**: El plano de gestión cubre las tareas estáticas que se manejan mejor fuera de los planos de aplicación, de control y de datos. Entre los ejemplos se incluyen la gestión de las relaciones comerciales entre el proveedor y el cliente, la asignación de recursos a los clientes, la configuración del equipo físico, la coordinación de la accesibilidad y las credenciales entre las entidades lógicas y físicas o la configuración de arranque de los sistemas. Cada entidad comercial tiene sus propias entidades de gestión. Uno de los objetivos de SDN es incluir muchas tareas de gestión de las redes anteriores en el CDPI [3].

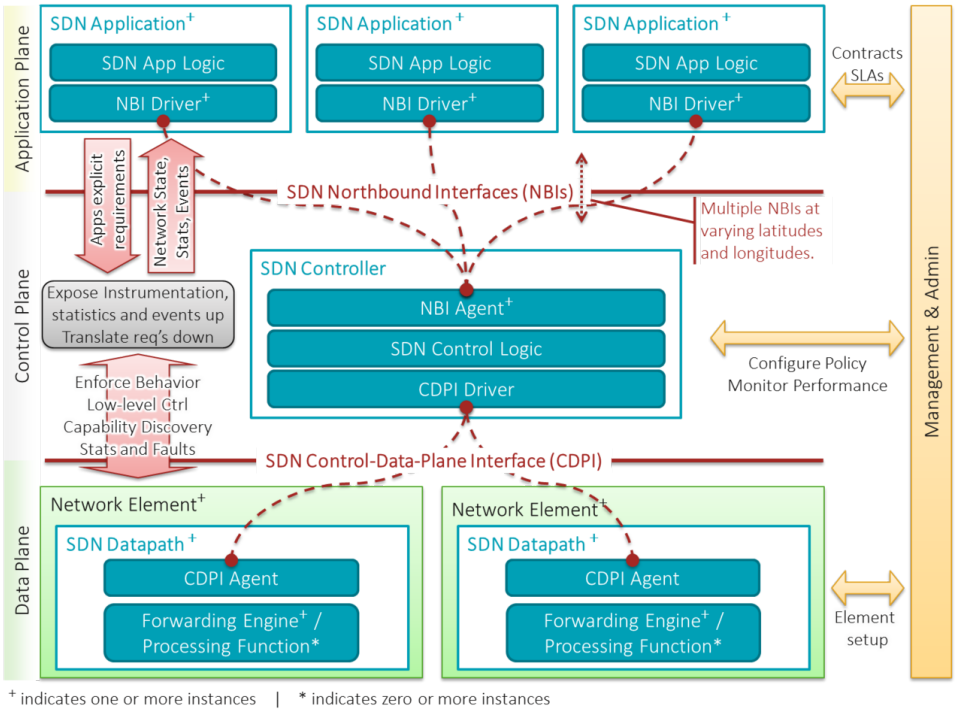


Figura 2.2: Arquitectura SDN detallada [3]

2.1.1.2 Principios en los que se basa la arquitectura SDN

El objetivo de SDN es ofrecer interfaces de código abierto que permitan el desarrollo de *software* capaz de controlar la conectividad proporcionada por un conjunto de recursos de red y el flujo de tráfico de red que los atraviesa, junto con la posible inspección y modificación del tráfico que puede ser llevada a cabo dentro de la red. La arquitectura de SDN se basa en tres principios estructurales:

- **Separación entre procesamiento y encaminamiento del trafico y el control**: El objetivo de este principio es permitir el despliegue independiente de entidades de reenvío y procesamiento de tráfico y entidades de control. Esta separación es un prerrequisito necesario para llevar a cabo un control centralizado y, además, permite optimizar de manera independiente la plataforma tecnológica y los ciclos de vida del *software*. Este desacoplo se ve reflejado en la arquitectura SDN mediante la figura del controlador SDN, que se encarga de controlar y gestionar un conjunto de recursos del plano de datos o capa de infraestructura.

- **Control lógicamente centralizado**: El término lógico quiere decir que el elemento de control se comporta como una entidad única, independientemente de que su implementación sea distribuida. El principio de control centralizado sostiene que los recursos pueden ser utilizados de manera más eficiente cuando se observan desde una perspectiva más amplia. Un controlador SDN puede orquestar recursos que a su vez generan entidades subordinadas, y por lo tanto ofrece un nivel de abstracción mayor a sus clientes.

- **Programabilidad de los servicios de red**: Este principio permite a los clientes intercambiar información con el controlador, tanto a través del descubrimiento o negociación previa al establecimiento del servicio, como durante el tiempo de vida del servicio de acuerdo con los cambios en las necesidades de los clientes o el estado de los recursos virtuales de éstos. Este intercambio propicia una mayor agilidad tanto a la hora de negociar de manera satisfactoria un servicio como a la hora de desplegarlo. Esto está relacionado con la habilidad de la tecnología SDN de aprovechar dinámicamente un amplio conjunto de recursos, o de crear nuevos recursos bajo demanda, como podrían ser las funciones de red virtuales [19].

2.1.1.3 Ventajas de la arquitectura SDN

SDN ofrece una red centralizada y programable, capaz de suministrar recursos de red de manera dinámica para satisfacer las necesidades cambiantes de los clientes. Además, proporciona distintas ventajas técnicas y económicas:

- **Directamente programable**: El desacoplo entre las funciones de reenvío y las funciones de control permite que la red sea reprogramada por diferentes herramientas de automatización.

- **Gestión centralizada**: La configuración de una unidad de control lógicamente centralizada permite mantener una visión global de la red.

- **Reducción del CAPEX (CAPital EXpediture)**: Potencialmente, SDN reduce la necesidad de comprar *hardware* específico de red, basado en ASIC. Además, permite establecer modelos de escalabilidad en los que los recursos se adquiren a medida que se necesitan.

- **Reducción del OPEX (OPerational EXpediture)**: La capacidad de automatizar las actualizaciones del *software* de red evita tener que reemplazar la infraestructura, o parte de ella, cuando las necesidades del negocio o de la red cambien. Asimismo, la automatización y la reprogramabilidad de la red permiten reducir el tiempo de gestión y el factor de error humano.

- **Agilidad y flexibilidad**: Las redes SDN permiten a las empresas desplegar nuevas aplicaciones, servicios e infraestructura de manera rápida y así adaptarse a los objetivos de negocio cambiantes.

2.1.1.4 Desventajas de la arquitectura SDN

Las redes SDN se enfrentan a problemas como la escalabilidad, la seguridad o la falta de cooperación en la industria.

- **Riesgos de seguridad debidos a la gestión centralizada**: La gestión centralizada hace que las unidades de control sean objetivo claro de los ataques dado que de ellas depende el funcionamiento de la red.
- **Cuello de botella en el controlador SDN**: Cuando existe una única unidad de control en una red con grandes cantidades de tráfico y numerosos dispositivos de red, puede producirse un cuello de botella en el controlador al no tener la capacidad de procesamiento suficiente para mantener la gestión de la red.
- **Inexistencia de estándares aceptados para las APIs NorthBound**: No disponer de un estándar para las APIs NorthBound aceptado de manera universal dificulta el desarrollo de las aplicaciones capaces de operar con diferentes controladores [17].

2.2 OpenFlow

El concepto original de OpenFlow surgió en la universidad de Stanford en 2008. Inicialmente, OpenFlow definió el protocolo de comunicación en entornos SDN que permite a la capa de control interactuar directamente con los dispositivos de red del plano de datos de la red SDN, dando lugar a una *Southbound API*. La versión 1.0 de la especificación de un switch OpenFlow fue lanzada en diciembre de 2009. Desde el comienzo, OpenFlow ha sido gestionado por la `Open Networking Foundation (ONF)`, una organización dirigida por los usuarios y dedicada a estándares abiertos y a la implementación de SDN [20].

Actualmente, la última especificación de un switch OpenFlow es la versión 1.5.1 [21], que data de abril de 2015. Sin embargo, en esta memoria se explican los conceptos más importantes de la **versión 1.3 de la especificación de un switch OpenFlow** [4]. Se ha tomado esta decisión porque los dispositivos del plano de datos, que se han considerado para desarrollar este proyecto, implementan dicha versión de la especifiación.

2.2.1 Componentes de un switch OpenFlow

Un switch OpenFlow está formado por una o más tablas de flujos (*flow tables*) y una tabla de grupos (*group table*), que llevan a cabo tareas de búsquedas y reenvío, y un `canal OpenFlow` a través del cual se comunica con un controlador externo, como se muestra en la Figura 2.3.

A través del protocolo OpenFlow, el controlador puede añadir, actualizar o borrar las entradas de las tablas de flujos, tanto de manera reactiva, como respuesta a los paquetes, como proactiva, anticipándose a la recepción del flujo. Cada entrada de una tabla de flujos contiene campos de `match`[1], contadores, y un conjunto de instrucciones que se aplican a los paquetes que cumplan los criterios de dicha entrada (Cuando un paqute cumple las condiciones de una entrada de la tabla de flujos se le denomina `match`). Estas tablas de flujos conforman el `pipeline` (Sección 2.2.2) del switch. Más concretamente, el concepto de `pipeline` se refiere al conjunto de tablas de flujos enlazadas que proporcionan *matching* [2], reenvío y la modificación de paquetes.

[1] Características que definenen un determinado flujo.
[2] Proceso en el cual se comparan los paquetes recibidos con las condiciones de las entradas de las tablas de flujos.

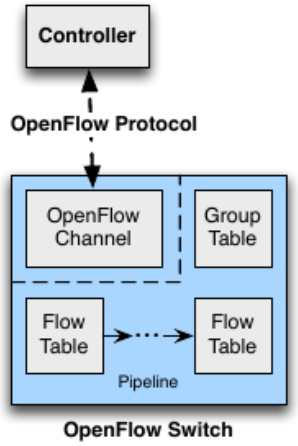


Figura 2.3: Componentes principales de un switch OpenFlow [4]

2.2.2 Pipeline

El `pipeline` OpenFlow de cualquier switch OpenFlow está formado por una o varias tablas de flujos, enumeradas de manera secuencial a partir del índice 0, y cada una de ellas contiene múltiples entradas. El `pipeline` define cómo van a interactuar los paquetes con dichas tablas de flujos, tal y como se puede observar en la Figura 2.4, donde se muestra el procesamiento al que se somete un paquete cuando entra en el. Cuando el switch OpenFlow únicamente dispone de una tabla de flujos, el procesamiento del `pipeline` se simplifica de manera substancial. El `pipeline` inicia el procesamiento del paquete al compararlo con las entradas de flujos de la tabla 0. Durante este procesamiento se comparas los campos de `match` del paquete con los de las entradas de flujos de la tabla. Cuando se produce un `match`, una coincidencia, con una de las entradas de la tabla, se ejecuta el conjunto de instrucciones asociado a esa entrada. Estas instrucciones pueden dirigir el paquete a otra tabla de flujos para que sea procesado, pero siempre a una tabla con un índice superior a la que lo está procesando, excepto la última tabla de flujos del `pipeline`, que no puede dirigir el paquete a ninguna otra tabla.

Cuando el paquete no es enviado a otra tabla, el procesamiento finaliza y el paquete es procesado junto con el conjunto de acciones que tiene asociado. Si el paquete no ha hecho `match` con ninguna entrada de las tablas se produce un `table miss`. Cuando esto ocurre, el paquete puede descartarse (*drop*), ser dirigido a otra tabla para que sea procesado o ser enviado al controlador. En esta versión de la especificación se implementa el *drop*, es decir, cuando no coincide con ninguna entrada de las tablas el paquete es descartado.

2.2.3 Tablas OpenFlow

La especificación de un switch OpenFlow define tres tipos de tablas para almacenar diferentes tipos de datos: tablas de flujos, de grupos y de métricas. A continuación, se explican más detalladamente cada uno de estos tipos.

2.2.3.1 Tablas de flujos

En esta tabla se almacena la información asociada a cada uno de los diferentes flujos que el switch debe procesar. Cada entrada de esta tabla queda definida unívocamente por los campos de `match` y la prioridad asignada. En la Figura 2.5 se muestran todos los campos que conforman cada una de las entradas que contiene este tipo de tabla:

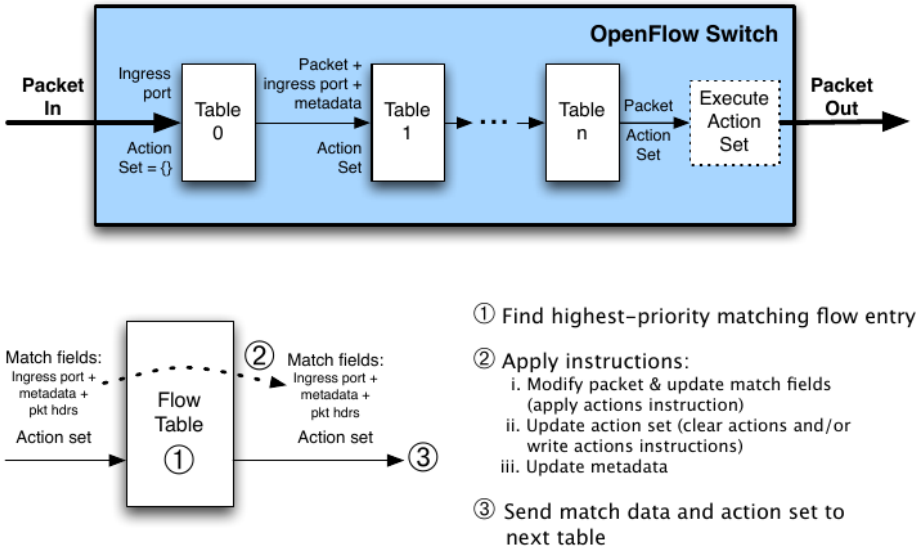


Figura 2.4: Procesamiento de un paquete a través del pipeline [4]

- **Match Fields**: Estos campos proporcionan la información que define un determinado flujo. Esta información incluye el puerto de entrada, campos de las cabeceras del paquete y, opcionalmente, metadatos especificados en una tabla anterior.

- **Priority**: Determina la prioridad de la entrada de flujo y se utiliza para ordenar las entradas dentro de la tabla. Las entradas con mayor prioridad se procesan antes.

- **Counters**: Los contadores se actualizan cuando se produce un `match` de un paquete con la entrada de flujo de la tabla. Se utilizan para proporcionar datos estadísticos.

- **Instructions**: Las instrucciones se utilizan para modificar el conjunto de acciones o el procesamiento del `pipeline`.

- **Timeouts**: Los temporizadores especifican el tiempo máximo de duración de una entrada o el tiempo máximo entre coincidencias con esa entrada antes de que caduque y sea eliminada.

- **Cookie**: dato configurado de manera opaca por el controlador. Puede ser utilizado por el controlador para filtrar estadísticas, modificar y eliminar los flujos.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Figura 2.5: Campos de la tabla de flujos [4]

Despues de conocer la información que contienen estas tablas y la estructura de sus entradas, en la Figura 2.6 se detalla el flujo que sigue un paquete al comenzar su procesamiento en la primera tabla del `pipeline`. El switch comienza a comparar los campos de `match` extraídos del paquete con los que aparecen en las entradas de la tabla de flujos. Un paquete hace `match` con una entrada de la tabla cuando los valores de los campos de `match` del paquete coinciden exactamente con los que aparecen en la entrada de la tabla.

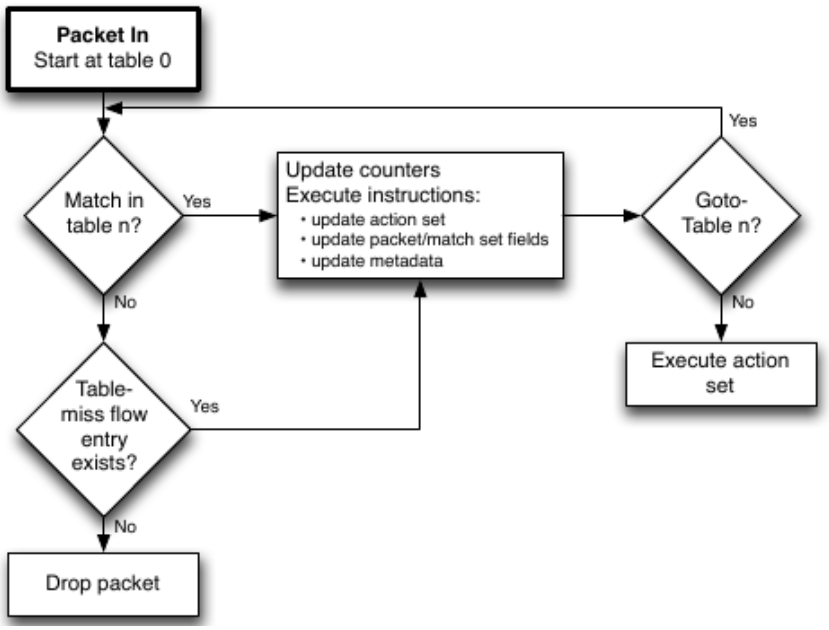


Figura 2.6: Diagrama de flujo del procesamiento de un paquete [4]

2.2.3.2 Tabla de grupos

Esta tabla permite implementar métodos adicionales de reenvío. En la Figura 2.7 se muestran los campos que definen cada una de las entradas de la tabla.

Group Identifier	Group Type	Counters	Action Buckets

Figura 2.7: Campos de la tabla de grupos [4]

- **Group Identifier**: Es un entero sin signo de 32 bit y permite identificar un grupo de manera única.
- **Group Type**: Indica el tipo de grupo.
  - All: Ejecuta todos los conjuntos de acciones del grupo. Se utiliza especialmente para tráfico *multicast* o *broadcast*.
  - Select: Ejecuta un conjunto de acciones del grupo.
  - Indirect: Ejecuta el conjunto de acciones definido en el grupo.
  - Fast failover: Ejecuta el primer conjunto de acciones que siga activo.
- **Counters**: Porporcionan datos estadísticos.
- **Action Buckets**: Se trata de una lista ordenada de grupos de acciones donde cada uno define una serie de acciones a ejecutar y los parámetros asociados.

2.2.3.3 Tabla de métricas

Las entradas de esta tabla definen métricas para cada flujo y permiten implementar operaciones de calidad de servicio como la limitación de la tasa de paquetes. Los campos que definen las entradas de la tabla de métricas se muestran en la Figura 2.8.

Meter Identifier	Meter Bands	Counters

Figura 2.8: Campos de la tabla de métricas [4]

- **Meter Identifier**: Permite identificar la métrica de manera unívoca.
- **Meter Bands**: Una lista de valores que especifican el rango de cada métrica y la forma de procesar los paquetes.
- **Counters**: Proporcionan datos estadísticos.

2.2.4 Mensajes OpenFlow

El protocolo OpenFlow define tres tipos de mensajes: controlador a switch, asíncronos y simétricos, cada uno de ellos con múltiples subtipos.

2.2.4.1 Mensajes controlador a switch

Estos mensajes son enviados desde el controlador y se utilizan para configurar u obtener información acerca del estado del dispositivo OpenFlow.

- **Features**: El controlador puede solicitar las prestaciones del switch mediante una petición (FEATURES_REQUEST). El switch debe responder especificando sus características (FEATURES_REPLY). Generalmente, este intercambio de mensajes se produce al establecer el canal OpenFlow.
- **Configuration**: Permite al controlador solicitar y configurar parámetros de configuración del switch. El switch únicamente responde cuando el controlador realiza una consulta.
- **Modify-State**: El controlador utiliza estos mensajes para gestionar el estado del switch. Su función principal es añadir, borrar o modificar las entradas de las tablas de flujo y grupos, y configurar las propiedades de los puertos del switch.
- **Read-State**: Estos mensajes son utilizados por el controlador para obtener información como la configuración actual, las estadísticas o las características del switch.
- **Packet-out**: El controlador utiliza estos mensajes para indicar al switch que envíe un paquete por un puerto determinado del `datapath`. El mensaje PACKET_OUT debe incluir el paquete completo o bien el identificador del *buffer* del switch donde ha guardado dicho paquete. Además, el paquete debe contener una lista ordenada con las acciones que se deben aplicar al paquete. En caso de que esta lista esté vacía el paquete se descarta.
- **Barrier**: Estos mensajes son utilizados por el controlador para garantizar que se han cumplido las dependencias de los mensajes o para recibir notificaciones de operaciones completadas.
- **Role-Request**: El controlador utiliza estos mensajes para configurar el rol de su canal OpenFlow o para consultarlo. Es útil cuando el switch se conecta a varios controladores.
- **Asynchronous-Configuration**: El controlador utiliza estos mensajes para configurar un filtro adicional para los mensajes asíncronos que quiere recibir a través de su canal OpenFlow, o bien para consultar el filtro existente.

2.2.4.2 Mensajes asíncronos

Los mensajes asíncronos son aquellos que el switch envía al controlador sin que este haya hecho ninguna petición previa. Los switches envían mensajes asíncronos para indicar que han recibido un paquete, que ha cambiado el estado del switch o se ha producido un error. A continuación, se describen los 4 tipos principales de mensajes asíncronos:

- **Packet-in**: Mediante este mensaje el switch transfiere el control del paquete al controlador. El switch puede configurarse para que al producirse un evento de PACKET_IN, almacene el paquete que lo ha generado y el mensaje PACKET_IN enviado al controlador contenga únicamente un número de bytes determinado del paquete, o bien para que envíe el paquete completo al controlador.
- **Flow-removed**: Estos mensajes indican al controlado que se ha eliminado una entrada de la tabla de flujos. Para que el switch genere este mensaje, la entrada de la tabla de flujos debe tener activado el flag OFPFF_SEND_FLOW_REM.
- **Port-status**: Informan al controlador acerca de un cambio en un puerto. Se espera que el switch envíe estos mensajes cuando la configuración o el estado de un puerto haya cambiado.
- **Error**: El switch utiliza estos mensajes para notificar al controlador de algún error que se ha producido.

2.2.4.3 Mensajes simétricos

Los mensajes simétricos se envían en ambas direcciones sin necesidad de que el destinatario haya enviado una solicitud previamente.

- **Hello**: El intercambio de mensajes HELLO se produce entre el switch y el controlador durante el inicio de la conexión.
- **Echo**: Se utilizan principalmente para comprobar si la conexión entre el controlador y el switch sigue activa, aunque también pueden emplearse para medir la latencia o el ancho de banda.
- **Experimenter**: Estos mensajes brindan la posibilidad a los switches OpenFlow de ofrecer funcionalidades adicionales [4].

2.3 Basic OpenFlow Userspace Software Switch (BOFUSS)

El BOFUSS, o también conocido como OfSoftSwitch13, es un *software-switch* de código abierto programado en C y C++ y desarrollado en el centro de investigación CPqD [15]. El BOFUSS implementa la versión 1.3 de OpenFlow y está basado en una desarrollo previo realizado por Ericsson Research TrafficLab [22] en el que se implementaba la versión 1.1 de OpenFlow.

2.3.1 Arquitectura del switch BOFUSS

La arquitectura del switch BOFUSS convierte a este `software-switch` en una opción muy interesante para prototipar y llevar a cabo proyectos de investigación. A continuación, se describen las partes más importantes que consituyen estructura, y que pueden observarse en la Figura 2.9:

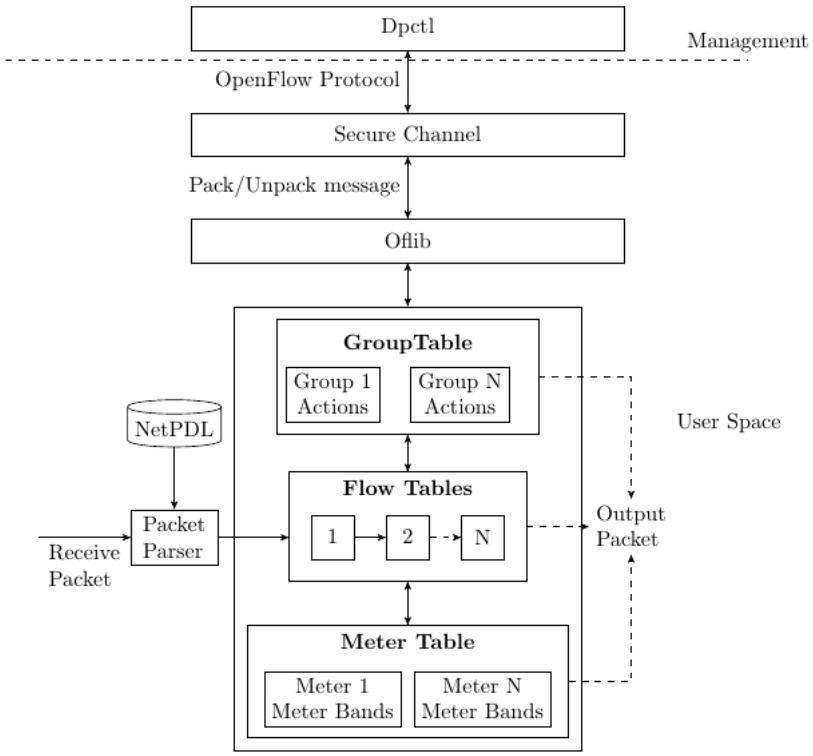


Figura 2.9: Arquitectura de BOFUSS [5]

- **Espacio de ejecución**: El switch BOFUSS realiza el procesamiento de los paquetes en el espacio de usuario. Esto es posible gracias a los *packet socket*[3] de Linux, que permiten que el switch reciba y envíe paquetes completos de cualquier protocolo sin cambiar sus cabeceras [5].

- **Puertos**: Los puertos OpenFlow son los canales de entrada y salida de los paquetes de red al `pipeline`[4] del switch OpenFlow. Una instancia del *software-switch* corriendo en una maquina puede utilizar los puertos físicos o las interfaces virtuales como puertos. Las funciones de los puertos del switch no se limitan a enviar y recibir paquetes, sino que existen un conjunto de responsabilidades asociadas con el protocolo OpenFlow y con el `pipeline`:

  - OpenFlow permite cierto control sobre el comportamiento de los puertos. Los mensajes de modificación de los puertos permiten configurarlos para que descarten todos los paquetes recibidos o reenviados, para que no generen mensajes PACKET_IN a causa de recibir flujos desconocidos o modificar su estado, entre otras opciones. Los puertos deben cambiar su comportamiento en función de la configuración enviada por el controlador.

  - Los puertos OpenFlow mantienen el estado actual de los puertos físicos. Esta información no puede ser configurada por un controlador OpenFlow, pero el switch debe informar al plano de control acerca de los cambios de estado de los enlaces. Los puertos monitorizan el estado de los puertos y actualizan su información en función de los cambios.

  - Un controlador OpenFlow puede solicitar al switch la descripción de los puertos.

  - Los puertos deben actualizar sus contadores de paquetes y los de las colas.

[3] http://man7.org/linux/man-pages/man7/packet.7.html
[4] Así es como se denomina al procesado de los paquetes a través de las tablas de flujos del datapath.

- **Analizador de paquetes**: Para incorporar un nuevo protocolo a OpenFlow es necesario añadir código específico para extraer los nuevos campos introducidos. Cada protocolo puede tener métodos singulares y complejos para extraer estos campos. De hecho, los campos variables como las opciones IP requieren una inspección a fondo de los paquetes. Por ello, la implementación del analizador de paquetes debe ser flexible y fácil de ampliar. La Figura 2.10 muestra la aquitectura implementada para el analizador de paquetes en el datapath del switch BOFUSS. Antes de llegar al `pipeline`, un paquete atraviesa el analizador de paquetes, donde se obtienen los campos de la cabecera. El analizador de paquetes utiliza la librería `NetBee` [23] para obtener estos campos. Esta librería utiliza la base de datos `NetPDL` [24], que contiene la definición de las cabeceras soportadas por OpenFlow 1.3 en formato XML (*eXtensible Markup Language*). La base de datos `NetPDL` permite añadir nuevos campos a OpenFlow de manera sencilla y es especialmente útil para las cabeceras variables. Para realizar el análisis de paquetes, `NetBee` en primer lugar carga en la memoria de acceso aleatorio (RAM) las especificaciones de los protocolos definidas por `NetPDL`. A continuación, los paquetes recibidos son decodificados de acuerdo con las descripciones proporcionadas por `NetPDL`, y la información obtenida se almacena en un árbol de protocolos. Una vez obtenida la información, el módulo `nbee_link` convierte las cabeceras extraídas de los paquetes en estructuras de `matches` que se comparan con las estructuras de `matches` de las entradas de las tablas de flujos del `pipeline`.

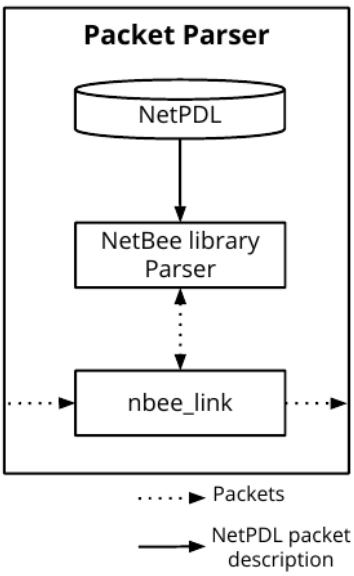


Figura 2.10: Componentes del analizador de paquetes [6]

- **Tablas de flujos**: Representan el corazón de la arquitectura de un switch OpenFlow, dado que en ellas se inicia el procesamiento del *pipeline* de OpenFlow. Esta tipo de tabla almacena las entradas de flujos mediante una lista enlazada. Las entradas de la tabla están ordenadas por prioridad, siendo las entradas de mayor prioridad las primeras en consultarse al realizarse el proceso de *matching* de los paquetes. Las funciones de la tablas de flujos dentro del switch OpenFlow son:
  - Implementar una acción por defecto en caso de que no se produzca ninguna coincidencia con los flujos almacenados en la tabla. En OpenFlow 1.3 la acción por defecto es descartar los paquetes.
  - Manejar los mensajes FLOW_MOD enviados por el controlador. Estos mensajes pueden añadir o eliminar entradas de la tabla, o bien cambiar el conjunto de instrucciones para alguna de las entradas de la tabla.

  - Las tablas de flujos deben permitir ser reconfiguradas por el controlador. Además, deben ser capaces de comunicar las propiedades que soportan, como por ejemplo el tipo de instrucciones o los campos de `match` permitidos en las tablas. Cada tabla posee un identificador para distinguirse de las demás, convirtiéndose en un parámetro esencial porque es indispensable especificar el identificador de la tabla a la hora de añadir un nuevo flujo. También debe tenerse en cuenta el número máximo de flujos que puede albergar una tabla para evitar problemas de escalabilidad.
  - La comparación de los campos de `match` del flujo con las entradas de la tabla debe hacerse nada más recibir el paquete. En caso de que se produzca un `match`, el switch ejecuta el conjunto de instruciones asociado a la entrada de flujo. Esta es la actividad más común de las tablas de flujo.
  - Mantener las estadísticas de las tablas en referencia al número de entradas de flujo activas, número de búsquedas realizadas y números de paquetes que se han asociado, es decir, han hecho *match*, a cada una de las entradas de la tabla.

- **Tabla de grupo**: La tabla de grupo mejora las opciones de reenvío. Los paquetes alcanzan la tabla de grupo una vez han hecho `match` con alguna de las entradas de las tablas de flujos que contenga un acción de grupo. Cada entrada de esta tabla contiene un identificador, un tipo, contadores y conjuntos de acciones (*action buckets*). Los conjuntos de acciones son listas ordenadas de acciones que deben ser ejecutadas de acuerdo con el tipo de grupo. La Figura 2.11 representa una tabla de grupo que contiene los tipos de grupo *All*, *Indirect* y *Fast Failover*. La disposición de los tipos de grupo específicos es importante, porque define las atribuciones de la tabla de grupo, como se muestra en las siguientes responsabilidades:
  - La tabla de Grupo tiene que garantizar las restricciones según el tipo de grupo. Por ejemplo, los grupos *Indirect* soportan un único conjunto de acciones.
  - La tabla de grupos debe manejar mensajes de modificación y llevar a cabo comprobaciones de consistencia en caso de que se produzcan encadenamientos de grupo. Los grupos encadenados apuntan a otros grupos y pueden causar bucles que deben ser evitados.
  - Los grupos *Fast Failover* necesitan que los puertos del switch y los conjuntos de grupos sean monitorizados para detectar cambios de estado. Por esta razon, la tabla de grupos es responsable de comprobar el tiempo de vida del conjunto a la hora de escoger el primer conjunto de acciones con vida.
  - Una tabla de grupos que soporta el tipo de grupo *Select* debe implementar un algoritmo de planificación (*scheduling*) para elegir qué conjunto de acciones se debe aplicar al paquete.

- **Tabla de métricas**: Su función es llevar a cabo operaciones de calidad de servicio (*Quality of Service*). Las métricas de cada flujo se asocian a las entradas de flujos a través de la instrucción `Meter`. Una entrada de métricas está formada por un identificador, contadores y un conjunto o grupo de métricas. Las operaciones de QoS que se van a aplicar vienen definidas por los conjuntos de métricas. Un grupo de métricas debe tener un tipo y un valor para la tasa de bits, que representa el límite para aplicar la acción indicada por el tipo de medidor. La Figura 2.12 representa una tabla de métricas, con dos tipos de grupos de métricas. Dentro de los cometidos de las tablas de métricas se incluyen:
  - Creación, destrucción y modificación de las entradas de métricas.
  - Medir la tasa de paquetes que han hecho `match` desde los flujos que apuntan a la tabla de métricas.

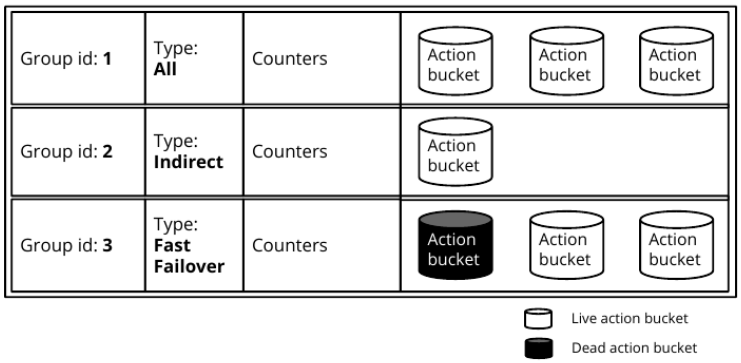


Figura 2.11: Tabla de grupo [6]

– Mantener y actualizar los contadores de las estadísticas de los paquetes procesados por una entrada.

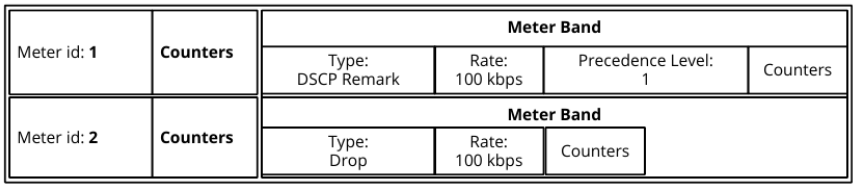


Figura 2.12: Tabla de métricas [6]

- **Oflib**: Los mensajes OpenFlow definidos por la especificación siguen un formato adecuado de transmisión a través de la red. Los mensajes son alineados a 8 *bytes*, así que, es posible que sea necesario añadir *bytes* de relleno para alcanzar la alineación correcta. Otro de los requisitos en el formato de los mensajes es el orden de *byte* (*endianness*[5]). Como los mensajes OpenFlow se transmiten a través de redes IP, estos mensajes deben ensamblarse siguiendo el formato `Big-Endian`. La arquitectura del procesador del equipo puede operar con distinto orden de byte. De hecho, los procesadores de Intel utilizan `Little-Endian`. Por lo tanto, para manejar y ensamblar los mensajes OpenFlow, es necesario convertirlos si se utiliza una arquitectura que no es `Big-Endian`. Por esta razón, es imprescindible disponer de una librería que se encargue de abstraer el orden de bytes utilizados y añadir los bytes de relleno necesarios para asegurar que los mensajes se han construido de manera correcta. La función principal de Oflib es convertir los mensajes OpenFlow del formato de red al formato utilizado en el equipo y viceversa. La biblioteca es responsable de:

  – Ofrecer para cada tipo de mensaje una función que lo empaquete y otra que permita desempaquetarlo (*pack/unpack*). La función *pack* se encarga de convertir las estructuras internas al formato de red, mientras que la función *unpack* se encarga de convertir los mensajes recibidos al formato interno.

  – Maneja algunos mensajes de error de OpenFlow. La librería debe señalar como erróneos los mensajes con un tamaño incorrecto o con argumentos desconocidos.

- **Canal Seguro**: Se trata de un programa independiente y autónomo que se encarga de establecer una conexión entre el switch y el controlador. Esta separación con el `datapath` se debe a que OpenFlow no especifica un método de conexion, por lo tanto, las implementaciones son libres de

[5] https://lefunes.wordpress.com/2008/05/13/endianess-big-endian-y-little-endian/

elegir el protocolo de conexión, como por ejemplo TCP o SSL (*Secure Socket Layer*). A pesar de que aparezca "seguro" en el nombre, por el momento, este componente solo soporta el protocolo TCP. Las obligaciones del canal seguro son:

  - Establecer una conexión TCP entre el switch y el controlador.
  - Gestionar la negociación de la versión del protocolo.
  - Establecer múltiples conexiones de manera simultánea con un único controlador. Estas conexiones pueden utilizarse para mandar mensajes OpenFlow en paralelo o para crear canales para mensajes específicos.
  - Permitir al switch comunicarse con más de un controlador OpenFlow en caso de que sea necesario[6].

- **Gestión**: El switch incluye la herramienta `Dpctl` que permite monitorizar y gestionar el switch mediante línea de comandos. Mediante `Dpctl` puede consultarse y modificarse el estado actual de los switches, por ejemplo, es posible añadir nuevos flujos, consultar las estadísticas actuales de los flujos o comprobar el estado de los puertos.

Tras describir las piezas más importantes de la estructura del switch BOFUSS, en la Figura 2.13 se presenta una visión mas detallada de la configuración de su `datapath`, para intentar ofrecer mayor claridad acerca del procesamiento de los paquetes por parte del *software-switch*.

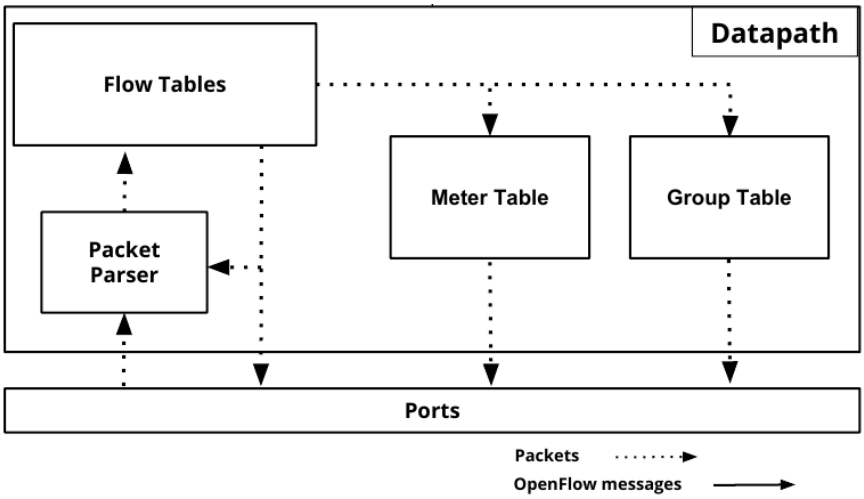


Figura 2.13: Arquitectura del datapath de BOFUSS [6]

2.4 Open vSwitch

El Open vSwitch, o también conocido como OvS, es un *software-switch* multicapa bajo licencia de código abierto Apache 2. El objetivo de este dispositivo *software* es implementar una plataforma de switch con una calidad de producción que soporte interfaces de gestión estándar y permita extender las funciones de reenvío y control. El OvS tiene la capacidad de funcionar como un switch virtual en entornos de máquinas virtuales (VM). Además, ha sido diseñado para poder funcionar de manera distribuida en varios servidores físicos y, por lo tanto, soporta múltiples tecnologías de virtualización basadas en Linux tales como `Xen/XenServer, KVM y VirtualBox`. La mayor parte de su código está escrito en C, por lo que su exportación a otros entornos es menos costosa.

El Open vSwitch cuenta con un módulo en *kernel* de Linux que ofrece un alto rendimiento en las tareas de reenvío. No obstante, también puede funcionar completamente en espacio de usuario sin hacer uso del módulo *kernel*, a cambio de perder parte de la mejora de rendimiento que aporta el módulo programado en *kernel*.

Los elementos principales de esta distribución son:

- **ovs-vswitchd**: Es un demonio que implementa el switch junto con un módulo en *kernel* de Linux para la conmutación basada en flujos.
- **ovsdb-server**: Es un server de bases de dato ligero al cual el ovs-vswitchd manda solicitudes para obtener su configuración.
- **ovs-dpctl**: Es una herramienta para configurar el modulo switch en *kernel*.
- **ovs-vsctl**: Es una utilidad para hacer peticiones y actualizar la configuración del ovs-vswitchd.
- **ovs-appctl**: Es una herramienta que envía comandos para lanzar demonios Open vSwitch.
- **ovs-ofctl**: Es una herramienta para enviar peticiones y controlar switches OpenFlow y controladores.

En la Figura 2.14 se puede observar un esquema de la interacción de los principales elementos que conforman el OvS.

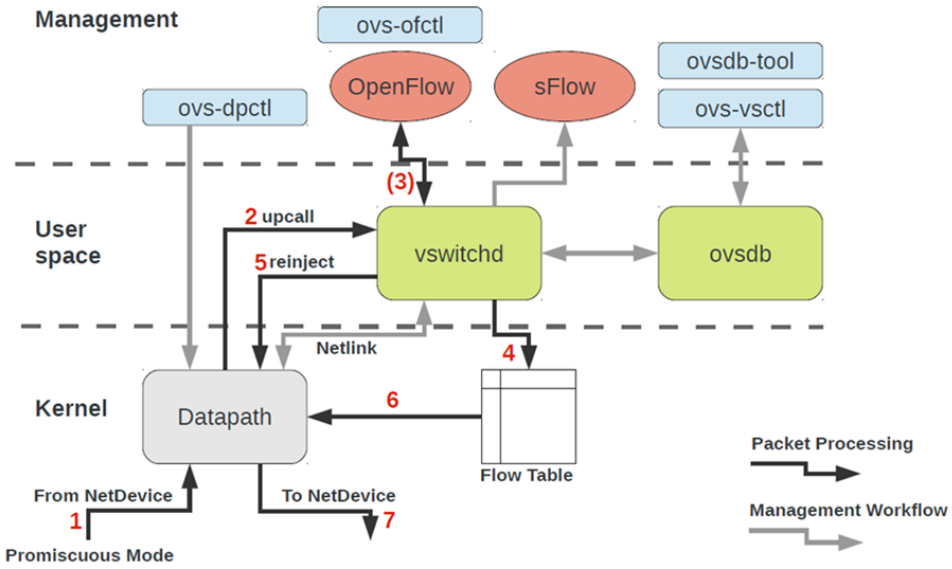


Figura 2.14: Arquitectura Open vSwitch [7]

2.5 Amaru

Amaru es un protocolo que permite establecer el control *in-band* en entornos SDN. Ofrece un despliegue automático de gran velocidad y múltiples caminos para mejorar la resiliencia del entorno, mientras reduce el número de entradas necesarias en las tablas para prevenir problemas de escalabilidad [8].

Para lograr satisfacer las afirmaciones anteriores, Amaru explora todos los caminos posibles entre cada dispositivo de red y el nodo raíz, siendo este el nodo conectado directamente al controlador SDN.

El hecho de tener constancia de todos los caminos disponibles garantiza la disponibilidad de rutas de respaldo en caso de la caída de un enlace o de un nodo.

Los diferentes caminos obtenidos durante la exploración quedan definidos por un **sistema de etiquetado jerárquico**. Este sistema consiste en asignar etiquetas de manera jerárquica comenzando por el nodo raíz. Las etiquetas aumentan un nivel por cada salto en el camino, tal y como queda ejemplificado en la Figura 2.15. Estas etiquetas se calculan para enmascarar las direcciones MAC (*Media Access Control*) reales de los dispositivos de red. A las nuevas direcciones obtenidas se las denominan MACs Amaru (**AMAC**s) y se distinguen de las MACs tradicionales mediante la activación del bit U/L (*unique/locally administered*) de la cabecera Ethernet. Este método de implementación de etiquetas permite mantener intacta la cabecera Ethernet tradicional.

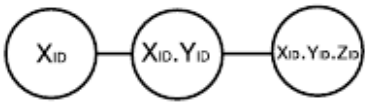


Figura 2.15: Etiquetado jerárquico en Amaru [8]

La exploración de los caminos comienza cuando un nodo descubre que está directamente conectado con el controlador. En se momento se autoproclama nodo raíz y envía una trama Amaru (*Amaru Frame o AFrame* Figura 2.16) por todos sus puertos. El proceso de exploración se implementa de manera distribuida, es decir, el nodo raíz obtiene la primera AMAC, que contiene un único nivel, y continuación, propaga por cada uno de sus puertos la AMAC de siguiente nivel formada por su AMAC más un identificador que el nodo elija, con la condición de que este sea único y consistente, es decir, siempre el mismo para cada nodo hijo. Cuando el nodo hijo recibe la AMAC, la asocia al puerto por el que la ha recibido y la propaga por todos sus puertos, exceptuando el de entrada, añadiéndole un nivel más a la dirección AMAC, según se ha indicado anteriormente.

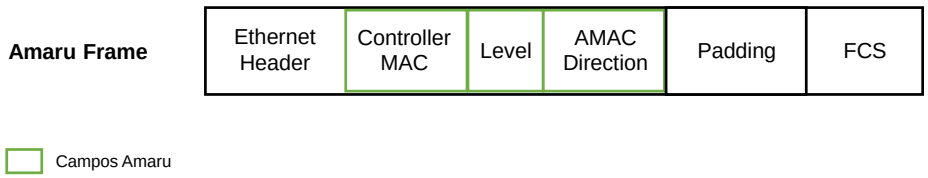


Figura 2.16: Trama Amaru (AFrame) [8]

En la Figura 2.17 se muestra cómo se llevaría a cabo la propagación inicial de las AMACs. El nodo s1, que es el nodo raíz, propaga la AMAC 1.7 al nodo s2 y la 1.2 al s4. Cuando estos dos nodos reciben la AMAC, la asocian al puerto de entrada y propagan la AMAC de siguiente nivel por todos sus puertos, excepto por el de entrada. En el caso del nodo s2, propaga la AMAC 1.7.3 hacia el nodo s3 y la AMAC 1.7.6 hacia el s5 y así sucesivamente. Para evitar bucles, los nodos descartan las AMACs que sean hijas, es decir, las AMACs que contengan el prefijo que el nodo ha difundido. En este ejemplo, el nodo s2 descartaría una AMAC recibida que fuese 1.7.X.

El protocolo utiliza dos parámetros para restringir el número de caminos que puede aprender un nodo. Con ello pretende que se genere un número de caminos alternativos suficiente, utilizando el mínimo número de AMACs:

- **N**: Para indicar el número máximo de AMACs que un nodo puede aprender.
- **L**: Para indicar el número de campos del prefijo de una AMAC que deben ser distintos a las ya aprendidas para ser aceptada.

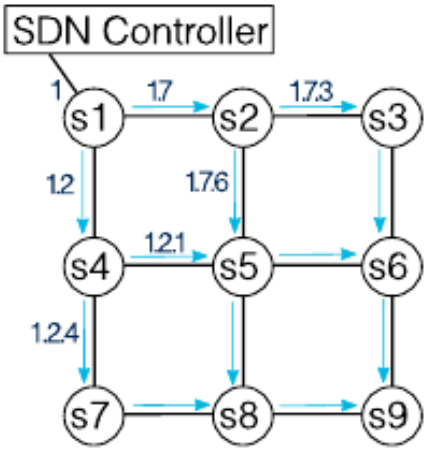


Figura 2.17: Propagación inicial de las AMACs [8]

Amaru implementa una reconfiguración de la red basada en los eventos producidos por la activación o desactivación de enlaces o nodos. El mecanismo más simple para llevar a cabo dicha implementación requiere que los nodos intercambien periódicamente mensajes HELLO con sus vecino, aunque podría implementarse a través de otros métodos como la monitorización de mensajes PORT_STATUS de OpenFlow. Gracias a la monitorización de estos eventos, Amaru tiene constancia de las AMACs que deben crearse, borrarse o desactivarse temporalmente.

- **Reconfiguración por evento provocado por la caída de un enlace**: Como las AMACs están asociadas a un puerto específico del nodo, cuando un enlace se desactiva, los nodos afectados deben desactivar las AMACs asociadas con ese puerto. El nodo debe decidir si deshabilitar la AMAC temporalmente o eliminar todo el árbol afectado. Esto último requiere enviar una AFrame a través de las ramas del árbol para indicar que debe borrarse la AMAC. En la Figura 2.18 se ilustra cómo el nodo s4 propagaría la eliminación de las AMACs afectadas por la caída del link entre el nodo s1 y el s4.
- **Reconfiguración por evento provocado por la caída de un nodo**: La caída de un nodo produce el mismo efecto que la caída de un enlace pero repetido n veces, siendo n el número de enlaces del nodo deshabilitado.

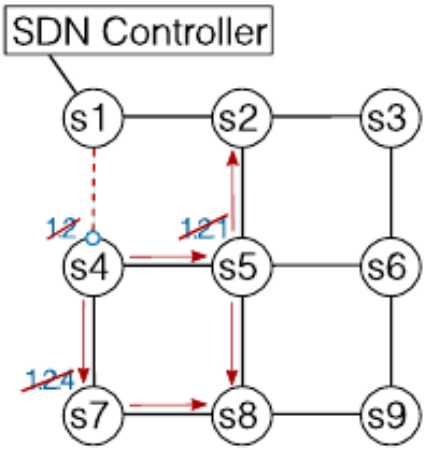


Figura 2.18: Reconfiguración de las AMACs originada por la caída de un enlace [8]

2.6 ONOS

ONOS (*Open Network Operating System*) es un proyecto de código abierto creado para acelerar la innovación en el campo de las Redes Definidas por *Software*, con el objetivo de reducir los costes que implican el despliegue y la gestión de las redes. Este proyecto se desarrolla en colaboración con múltiples proveedores de servicios como AT&T, NTT Communications o Verizon, numerosos proveedores de dispositivos de red como Cisco, Ericsson o Huawei, entre otros, y otros colaboradores. Además, está respaldado por la ONF (*Open Networking Foundation*)[6], un consorcio sin ánimo de lucro dirigido por operadores que impulsa la transformación de la infraestructura de red y los modelos de negocio de los operadores.

ONOS está concebido como un sistema operativo de red completo, ofreciendo funcionalidades que van más allá de un controlador SDN. Proporciona, herramientas como APIs que aportan abstracción para el desarrollo de aplicaciones SDN, o APIs para administrar, monitorizar y programar dispositivos de red. Además, proporciona vistualización, aislamiento, acceso seguro y abstracción sobre los recursos de red gestionados por el sistema operativo de la red. ONOS es capaz de multiplexar los recursos *hardware* y *software* entre las aplicaciones SDN. Asimismo, permite configurar políticas de red basadas en la intención de las aplicaciones, es decir, aplicar políticas de red que permitan satisfacer los requisitos de las aplicaciones y procesar eventos de red.

Está diseñado para operar con dispositivos de red *whitebox*[7] con el objetivo de rebajar los costes asociados a las soluciones propietarias. ONOS posee una arquitectura flexible que permite que el nuevo *hardware* se integre fácilmente en el *framework* SDN [25]. Puede funcionar como un sistema distribuido a través de múltiples servidores, esto permite utilizar los recursos de CPU y memoria de múltiples equipos, al mismo tiempo que se proporciona resistencia frente al fallo de los servidores y permite, potencialmente, hacer cambios en el *hardware* y en el *software* sin interrumpir el tráfico de red.

ONOS proporciona el plano de control para las Redes Definidas por *Software*, gestionando los componentes de red, como switches y enlaces, y ejecutando el *software* necesario para proporcionar los servicios de comunicación a los *hosts* finales y a las redes vecinas [26].

En la Figura 2.19 se muestran las características clave de la arquitectura de ONOS:

- **Core distribuido**: El sistema operativo SDN está diseñado para funcionar en un *cluster* implementando los más altos estándares de agilidad, resiliencia, resistencia frente a fallos, alto rendimiento y alta escalabilidad.

- **Abstracción/APIs Northbound**: Proporciona servicios de configuración y gestión para los desarrolladores de las aplicaciones SDN. El marco de intención de las aplicaciones les permite especificar sus requisitos de red, facilitando el desarrollo de las mismas.

- **Abstracción/APIs Southbound**: Proporciona los complementos de los distintos protocolos de gestión y configuración para comunicarse con los dispositivos de red.

- **Modularidad software**: Se ha diseñado una arquitectura de *software* modular con el objetivo de aumentar la eficiencia del desarollo, del despliegue y del mantenimiento.

[6] https://opennetworking.org/

[7] Dispositivos de red construidos con *hardware* básico, que utilizan chipsets (ASICs) de proveedores conocidos y permiten ejecutar un sistema operativo de red elegido por el cliente.

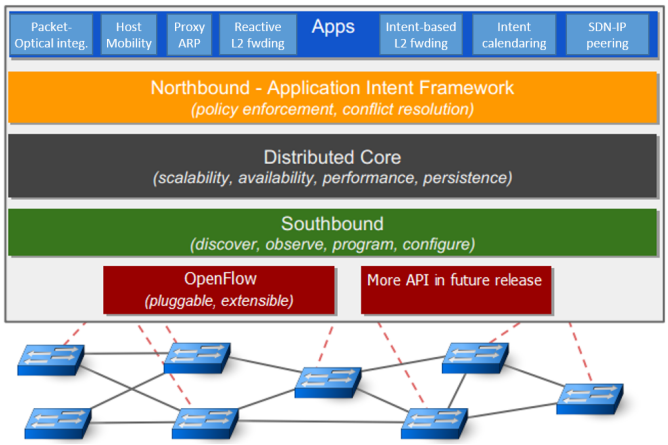


Figura 2.19: Arquitectura ONOS [9]

2.7 RYU

RYU es un *framework* de red basado en componentes y desarrollado por NTT (*Nippon Telegraph and Telephone*). Se trata de un *software* escrito en lenguaje Python cuya API permite a los desarrolladores crear nuevas aplicaciones de control y gestión de red de manera rápida y sencilla. Ryu es compatible con varios protocolos de gestión de dispositivos de red , siendo los más destacados OpenFlow, Netconf y OF-config. En cuanto a OpenFlow, es compatible con las versiones 1.0, 1.2, 1.3, 1.4, 1.5 y las extensiones Nicira [27].

Las aplicaciones RYU son entidades monohilo que implementan diferentes funcionalidades. Las aplicaciones utilizan eventos para mandarse mensajes asíncronos entre ellas. Cada aplicación Ryu cuenta con una cola FIFO para los posibles eventos que pueda recibir, permitiendo mantener el orden de los eventos recibidos. Cada aplicación Ryu dispone de un hilo destinado a procesar la cola FIFO. Para procesar los eventos, el hilo desencola un evento de la cola FIFO y llama al manejador del evento en cuestión. Mientras el manejador esté bloqueado, no se procesan otros eventos de la aplicación Ryu [28].

En la Figura 2.20 se observa como el *framework* SDN de RYU está dividido en dos capas. La capa superior representa la capa de aplicaciones y está compuesta por las aplicacionees que incorpora Ryu, las nuevas aplicaciones programadas y las APIs northbound, que permiten la interacción entre las aplicaciones empresariales o definidas por el usuarios con el *framework* de Ryu. La capa inferior, denominada capa de control, incorpora las bibliotecas necesarias para implementar los distintos protocolos de gestión de dispositivos de red que incorpora Ryu.

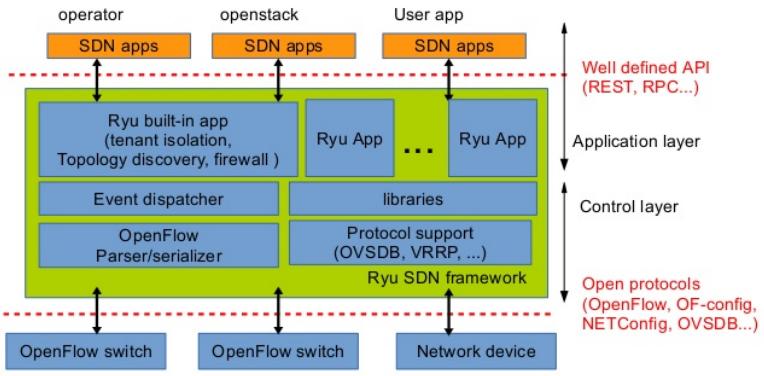


Figura 2.20: Arquitectura del framework SDN de RYU [10]

2.8 Network namespace

Los network namespaces son una herramienta proporcionada por el *kernel* de Linux para aislar recursos. Más concretamente, permiten que los procesos asociados a un network namespace tengan su propia pila de red (*network stack*) con su propia tabla de rutas, sus reglas de *firewall* y sus propios dispositivos de red, sin interferir con la pila del network namespace del sistema, conocido como root. Para la creación y el manejo de los network namespaces se utiliza la herramienta de linux iproute2[8] junto con el módulo netns[9]. Por convenio, un network namespace creado con esta herramienta es un objeto ubicado en **/var/run/netns/nombre** [29]. A continuación, se listan los principales comandos utilizados para gestionar los network namespaces:

Código 2.1: Comandos para la gestión de network namespaces

```
#!/bin/bash
#Crear un network namespace
ip netns add nombre_nns

#Borrar un network namespace
ip netns del nombre_nns

#Ejecutar un comando en el network namespace
ip netns exec nombre_nns comando

#Listar los network namespaces existentes
ip netns list
```

2.9 Interfaces Veth (Virtual Ethernet Device)

Los dispositivos veth son interfaces Ethernet virtuales que se crean como una pareja de interfaces interconectadas. Este hecho implica que los paquetes que se transmiten a través de una de las interfaces, inmediatamente son recibidos por la otra interfaz de la pareja. Otra característica importante de este tipo de dispositivos es que si uno de los los miembros de la dupla cambia su estado a *down*, el otro también lo hace.

Las interfaces veth se utilizan para conectar diferentes network namespaces. El proceso de conexión consiste en asignar cada miembro de la pareja a un network namespace diferente. De este modo se establece la comunicación entre los network namespaces [30]. A continuación, se listan los principales comandos utilizados para gestionar los dispositivos veth:

Código 2.2: Comandos para la gestión de las interfaces veth

```
#Crear un par veth
ip link add [nombre_veth1] type veth peer name [nombre_veth2]

#Borrar un par veth
ip link del [nombre_veth1|2]

#Asignar un extremo del par a un network namespace
ip link set [nombre_nns] netns [nombre_veth1|2]
```

[8] https://github.com/shemminger/iproute2
[9] https://man7.org/linux/man-pages/man8/ip-netns.8.html

Capítulo 3

Análisis y diseño

Las cosas no se hacen siguiendo caminos distintos para que no sean iguales, sino para que sean mejores.

Elon Musk

A lo largo de este capítulo se detallan las decisiones tomadas en relación con los objetivos del presente TFM, definidos en el primer capítulo de esta memoria (Sección 1.2), que marcan la línea de desarrollo explicada detalladamente en el Capítulo 4. Además, se describe la metodología de trabajo que se espera seguir durante el desarrolo del proyecto.

Las decisiones tomadas aclaran las siguientes incógnitas:

- Por qué utilizar el protocolo Amaru para implementar el canal de control *in-band*.
- Qué *software-switch* utilizar para llevar a cabo la implementación del protocolo Amaru.
- Qué controlador SDN utilizar durante el desarrollo.

3.1 Elección del protocolo Amaru

La implementación del protocolo Amaru como mecanismo para ofrecer el modo de control *in-band* resulta muy interesante. Haciendo referencia a la descripción del protocolo que se ofrece en la Sección 2.5, los principales motivos por los que se ha decidido llevar a cabo la implementación del protocolo Amaru son:

- Al tratarse de un protocolo de exploración, ofrece un enfoque diferente a la norma más extendida perpetuada por los protocolos basados en el estado de los enlaces.
- Ofrece mayor robustez al entorno que lo implementa gracias a la resistencia ante caídas de los enlaces que transportan mensajes de control.
- La estrategia de exploración permite reducir el tiempo de convergencia y el número de paquetes intercambiados.
- El protocolo Amaru ha sido comparado con otros protocolos similares como OSPF o RSTP, obteniendo frente a estos resultados satisfactorios en cuanto a tiempo de convergencia y número de paquetes necesarios para llevar a cabo la exploración [8].

- La posibilidad que ofrece Amaru de disponer de múltiples caminos hasta el controlador, junto con la reconfiguración del puerto de control en base a estos caminos, ofrecen mayor seguridad al canal de control gracias a la capacidad de reconfigurar en caliente el puerto de control.
- El protocolo ha sido desarrollado por el grupo de investigación GIST - Netserv de la Universidad de Alcalá.

3.2 Elección del software-switch

Para el desarrollo del proyecto se han contemplado dos *software-switches*, el el BOFUSS (Sección 2.3) y el Open vSwitch (Sección 2.4). Una vez analizadas ambas opciones, se ha optado por emplear el BOFUSS para llevar a cabo la implementación del protocolo Amaru. Los motivos que han impulsado a tomar esta decisión son los siguientes:

- Uno de los principales motivos es que el switch BOFUSS implementa en espacio de usuario la lógica que procesa los paquetes. Este hecho facilita enormemente la tarea de implementar el protocolo Amaru.
- El hecho de estar programado en C y C++ facilita la incorporación de nuevas funcionalidades.
- Existe una implementación básica de Amaru desarrolada en el switch BOFUSS [31]. Esta implementación supone una gran ventaja a la hora de implementar las funcionalidades que requiere este proyecto.
- La sencillez de la estructura del switch BOFUSS facilita las labores de prototipado, permitiendo implementar nuevas funcionalidades de una manera rápida y sencilla, al mismo tiempo que agiliza los experimentos.

3.3 Elección del controlador

Las dos opciones que se han tenido en cuenta para utilizar como controlador SDN son ONOS (Sección 2.6) y RYU (Sección 2.7). En este proyecto, el controlador debe hacer uso únicamente de una aplicación de reenvío (*forwarding*) y no está previsto implementar ninguna aplicación en el controlador durante el desarrollo del proyecto.

Tras analizar ambas opciones, se ha decidido utilizar principalmente ONOS aunque no se descarta la utilización de RYU para realizar pruebas puntuales de funcionamiento. A pesar de que la instalación de ONOS es más complicada, se ha optado por esta opción porque su uso está orientado al ámbito de los operadores de redes, por lo que es más estable y ofrece una interacción más amigable al contar con una interfaz gráfica. A través de esta interfaz es posible gestionar las aplicaciones del controlador, observar los dispositivos conectados y las topologías que conforman, además de consultar las características y estadísticas de los dispositivos.

3.4 Análisis del funcionamiento del switch BOFUSS

En este apartado se analiza la implementación del centro de investigación CPqD [15] del switch BOFUSS, descrito en la Sección 2.3.1, con el objetivo de obtener una visión global del funcionamiento del *software-switch* y establecer una línea de trabajo que permita implementar las modificaciones necesarias para alcanzar los objetivos del TFM.

En la implementación destacan dos bloques llamados `ofdatapath` y `ofprotocol`, que ejecutados conjuntamente dan lugar al switch BOFUSS.

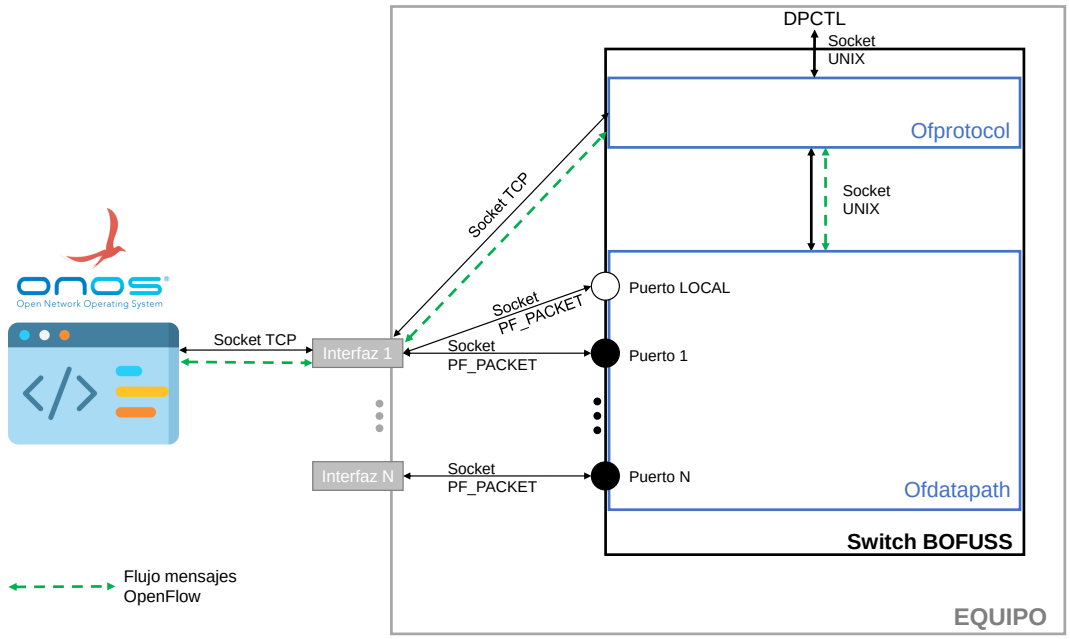


Figura 3.1: Diagrama de conexiones del switch BOFUSS

- **Ofdatapath**: Este módulo implementa las funcionalidades de un switch de capa 2, así como el datapath OpenFlow de la arquitectura de BOFUSS. Monitoriza permanentemente los puertos que tiene configurados y comprueba si han recibido paquetes. Cuando los puertos reciben datos, los paquetes son procesados por el `pipeline` del switch (Sección 2.2.2), que determina si los paquetes se encaminan a través de alguno de los puertos del switch o se descartan. Cuando el switch está operando en el modo de control *in-band* el puerto que se utiliza como puerto de control se configura en función de cómo se haya definido la opción `--local-port` al ejecutar el módulo `ofdatapath` (Sección 3.4.1). Si no se especifica ninguna interfaz como puerto local, el switch crea una interfaz `tap` [32], que es un dispositivo ethernet virtual, y la incorpora como puerto local. En el caso contrario, cuando se especifica una interfaz en la opción `--local-port`, el switch modifica la dirección MAC de la interfaz en función del ID[1] del switch e incorpora dicha interfaz como puerto local, dando lugar a un esquema como el que se muestra en la Figura 3.1.

- **Ofprotocol**: Este módulo implementa el canal seguro que se encarga de gestionar la conexión OpenFlow entre el switch y el controlador, tal y como se describe en la Sección 2.3.1. De esta manera, este bloque actúa como enlace entre el bloque `ofdatapath` y el controlador. En la figura anterior se puede apreciar que el módulo `ofprotocol` define 3 conexiones: la conexión DPCTL para recibir instrucciones a través de la línea de comandos, una conexión local mediante la que intercambia mensajes con el bloque `ofdatapth`, y una conexión remota a través de la cual se conecta con el controlador. Al actuar como intermediario entre el controlador y el módulo `ofdatapath`, analiza algunos mensajes que se intercambian, como son los mensajes PACKET_IN, FLOW_MOD o PORT_DESCRIPTION, con el fin de mantener el registro de la actividad y del estado de los puertos del switch.

[1] Si el switch tiene asignado, por ejemplo, el identificador 2, la MAC asignada a la interfaz del puerto de control será 00:00:00:00:00:02.

Según se ha comentado en la descripción del bloque `ofdatapath`, la interfaz a la que se asocia la conexión TCP, que el bloque `ofprotocol` establece con el controlador, depende del enrutamiento que el sistema lleve a cabo. Por lo tanto, el módulo establece la conexión del mismo modo para el control *in-band* como para el control *out-of-band*. La diferencia que existe en utilizar un modo de control u otro es que al utilizar el modo *in-band* se inicializa una lógica que permite realizar un procesado adicional de los mensajes PACKET_IN que el bloque `ofdatapath` envía al controlador y modificar el funcionamiento del switch en base a ese procesamiento para lograr encaminar correctamente las conexiones OpenFlow de otros dispositivos.

Es necesario recalcar que para utilizar el modo *out-of-band* es necesario añadir la opción `--out-of-band` al ejecutar el módulo `ofprotocol`(sección 3.4.2), dado que, por defecto, el switch BOFUSS configura el modo de control *in-band*.

La Figura 3.1 muestra las principales conexiones de los dos bloques que conforman el switch BOFUSS. Esta versión del switch no configura ni implementa realmente un puerto de control a través del que se establece la conexión OpenFlow, sino que los mensajes se transmiten a través de la interfaz por la que se alcanza el controlador según la tabla de rutas del sistema. Por lo tanto, y según se indica en las instrucciones de instalación del switch, para que el switch funcione correctamente en el modo *in-band*, es necesario configurar previamente dicha interfaz para que tenga asignada la dirección IP que se utilizará para establecer la conexión OpenFlow con el controlador.

Los módulos se inician mediante los comandos `ofprotocol` y `ofdatapath`, que deben ejecutarse con permisos de superusuario. Los argumentos de estos comandos pueden consultarse mediante la herramienta `man`, una vez se haya instalado la implementación del switch BOFUSS. Para comprenderlos en mayor profundidad se analizan mediante unos ejemplos de uso real.

3.4.1 Ofdatapath

En la Figura 3.2 se muestra el comando mediante el cual se ejecuta el módulo `ofdatapath`, así como el significado de los principales argumentos que lo conforman. De entre todos los argumentos, es necesario destacar el socket de escucha *punix:/tmp/s1*. A través de este socket se intercambian los paquetes del protocolo OpenFlow, descritos en la Sección 2.2.4, con el bloque `ofprotocol`.

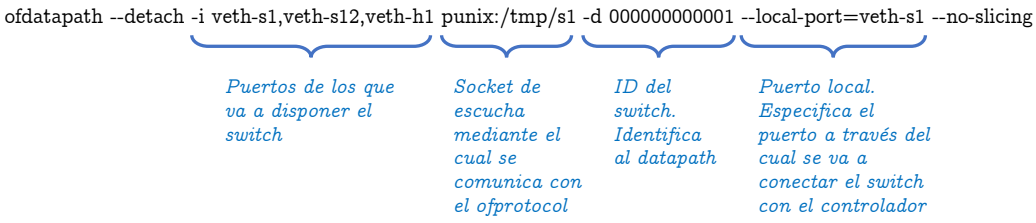


Figura 3.2: Comando para ejecutar el módulo ofdatapath

3.4.2 Ofprotocol

En la Figura 3.3 se detalla el comando a través del cual se ejecuta el módulo `ofprotocol` y se explican los argumentos más importantes. Es necesario resaltar la importancia de dos de los argumentos de este comando:

- **unix:tmp/s1**: Tal y como se ha mencionado en el comando `ofdatapath`, a través de este socket se intercambian los paquetes OpenFlow con dicho bloque.
- **tcp:10.0.0.88:6653**: Se utiliza para indicar el tipo de socket que se debe utilizar (TCP) para establecer la conexión con el controlador, cuya IP en el ejemplo es 10.0.0.88, y el puerto del controlador contra el que debe conectarse, en este caso el puerto 6653, que es el puerto por defecto de OpenFlow. A través de este socket se intercambian los paquetes OpenFlow con el controlador.

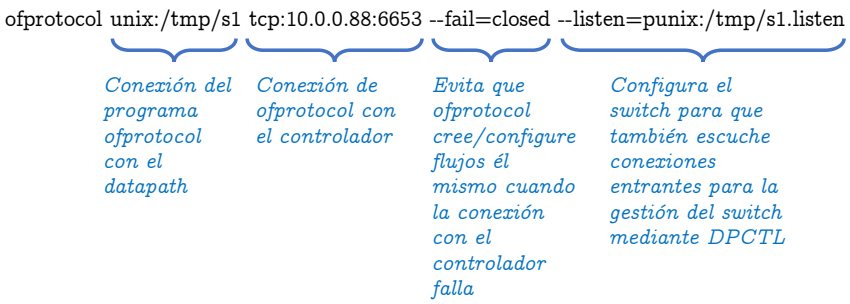


Figura 3.3: Comando para ejecutar el módulo ofprotocol

3.5 Diseño del método de trabajo con el switch BOFUSS

Una vez identificadas todas las piezas que constituyen el proyecto, el siguiente paso es especificar las modificaciones que deben implementarse en el switch BOFUSS para alcanzar los objetivos establecidos en la introducción de este TFM. No obstante, antes de continuar con la descripción de las modificaciones, es necesario definir el método de trabajo que se empleará durante el desarrollo del proyecto.

Tras haber analizado la arquitectura del switch y las distintas partes que lo componen en la Sección 3.4, es necesario comprender la ejecución del código de la implementación del *software-switch* para poder concretar de manera detallada las modificaciones que deben llevarse a cabo. Para ello, se ha establecido el flujo de trabajo que se muestra en la Figura 3.4. En primer lugar, se analiza y se depura el código del switch BOFUSS. Una vez se ha comprendido el funcionamiento, se plantean de manera detallada las modificaciones necesarias. A continuación, se modifica el código para implementar los cambios propuestos y, por último, se comprueba si la implementación funciona correctamente y satisface los objetivos establecidos. Si el funcionamiento de la implementación en conjunto con las modificaciones no es el adecuado o no cumple los objetivos, se regresa a la primera fase y vuelve a iniciarse el proceso.

3.5.1 Método para depurar el código del switch BOFUSS

Para depurar el código de *software-switch* se ha empleado la herramienta `Visual Studio Code` [33]. Se trata de una herramienta muy fácil de usar y muy versátil porque dispone de extensiones para casi cualquier lenguaje de programación. Dado que la implementación del switch BOFUSS está programada en lenguaje C, es necesario instalar la extensión `C/C++` de Microsoft [34] para habilitar la depuración

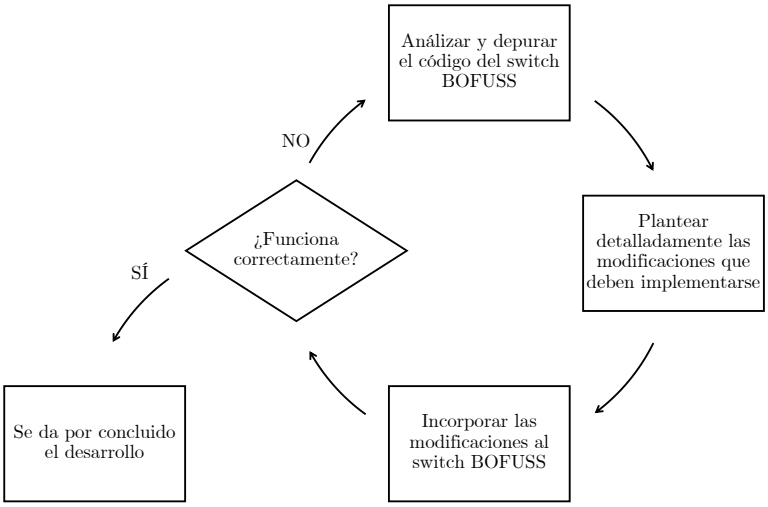


Figura 3.4: Diagrama de flujo del proceso de desarrollo

mediante el depurador `GDB (GNU Debugger)` [35] puesto que, tal y como se especifica en el anexo D, en el entorno en el que se desarrolla el proyecto se utiliza Ubuntu 16.04 LTS en su versión Desktop [36], una distribución GNU/Linux.

Antes de comenzar a depurar el código, es necesario llevar a cabo varias tareas. Los módulos `ofdatapath` y `ofprotocol` necesitan ejecutarse con privilegios de superusuario, por lo tanto, es necesario ejecutar la herramienta Visual Studio Code con los mismos privilegios. Para conceder privilegios de superusuario a la herramienta, es necesario que sea ejecutada mediante el siguiente comando:

Código 3.1: Comandos ejecutar VSC con privilegios de superusuario

```
1 sudo code --user-data-dir="~/.vscode-root"
```

Como se puede apreciar, es necesario especificar el directorio donde se guardarán los datos del usuario root, es decir, el usuario con privilegios de superusuario.

Una vez se ha ejecutado el programa con los privilegios necesarios, se accede mediante la herramienta Visual Studio Code a la ubicación del directorio raíz de la implementación del switch BOFUSS y se configuran los ajustes de depuración, es decir, se especifican los atributos y las opciones con las que se ejecutarán los módulos `ofdatapath` y `ofprotocol` cuando se inicie la depuración. En el bloque de Código 3.2 se muestra un ejemplo de configuración para cada uno de los módulos empleando los mismos atributos y opciones que aparecen en las secciones 3.4.1 y 3.4.2, donde se detalla el uso de los comandos con los que se ejecutan ambos bloques.

Código 3.2: Configuraciones para la depuración de los módulos ofdatapath y ofprotocol

```
1  {
2      "name": "(s1)ofprotocol",
3      "type": "cppdbg",
4      "request": "launch",
5      "program": "${workspaceFolder}/secchan/ofprotocol",
6      "args": "unix:/tmp/s1", "tcp:10.0.0.88:6653", "--fail=closed", "--listen=
       punix:/tmp/s1.listen",
7      "stopAtEntry": false,
8      "cwd": "${workspaceFolder}",
9      "environment": [],
10     "externalConsole": false,
```

```
11      "MIMode": "gdb",
12      "setupCommands": [
13      {
14          "description": "Ofprotocol",
15          "ignoreFailures": true
16      }
17      ]
18  },
19  {
20      "name": "(s1)ofdatapath",
21      "type": "cppdbg",
22      "request": "launch",
23      "program": "${workspaceFolder}/udatapath/ofdatapath",
24      "args": "-i", "veth-s1,veth-s12,veth-h1", "punix:/tmp/s1", "-d",
        "000000000001", "--local-port=veth-s1", "--no-slicing",
25      "stopAtEntry": false,
26      "cwd": "${workspaceFolder}",
27      "environment": [],
28      "externalConsole": false,
29      "MIMode": "gdb",
30      "setupCommands": [
31      {
32          "description": "Ofdatapath",
33          "ignoreFailures": true
34      }
35      ]
36  },
```

Para depurar simultáneamente ambos bloques, es necesario ejecutar dos instancias de la herramienta Visual Studio Code mediante el comando especificado en el blóque de códgo 3.1, especificando directorios diferentes para cada instancia.

Capítulo 4

Desarrollo

El éxito no es definitivo, el fracaso no es fatídico. Lo que cuenta es el valor para continuar.

Winston Churchill

A lo largo de este capítulo se detalla el desarrollo que se ha llevado a cabo para alcanzar y satisfacer los objetivos que abarca el proyecto. Este capítulo se distribuye en cuatro secciones: en la primera, se describen las modificaciones que se han implementado a lo largo del desarrollo del proyecto y en las tres siguientes, se detalla la implementación de cada una de las modificaciones que se han llevado a cabo.

4.1 Modificaciones implementadas durante el desarrollo del proyecto

Siguiendo la metodología presentada en el capítulo anterior, se ha depurado el código del switch BOFUSS para determinar cuáles son las modificaciones a implementar para alcanzar el objetivo final que consiste en implementar el modo de control *in-band* mediante el protocolo Amaru.

En la Figura 4.1 se puede observar la secuencia que siguen las modificaciones que se han implementado en el *software-switch*. En primer lugar, se detectó que la lógica que implementa el funcionamiento *in-band* del *software-switch* no se ejecutaba correctamente. Este problema estaba provocado por un desajuste en el tamaño del identificador del puerto local que impedía que los datos de este puerto se guardaran en la estructura en la que el módulo `ofprotocol` almacena los datos de los puertos del switch BOFUSS.

Una vez solucionado este problema, el siguiente paso fue implementar correctamente el modo de funcionamiento *in-band*. Para ello, era necesario que el switch fuera capaz de instalar ciertas reglas en su tabla de flujos de manera autónoma para poder encaminar el tráfico de las conexiones OpenFlow de otros dispositivos. Sin embargo, los mecanismos que generan los mensajes FLOW_MOD, que permiten instalar estas reglas, no funcionaban correctamente, ya que los paquetes originados eran erróneos y, por lo tanto, el módulo `ofdatapath` era incapaz de instalar las reglas, ya que etiquetaba estos paquetes como erróneos.

Tras modificar los mecanismos y conseguir que los mensajes FLOW_MOD instalasen correctamente las reglas, se observó que la lógica del bloque `ofprotocol` que analiza los mensajes PACKET_IN antes de ser enviados al controlador no funcionaba correctamente. Esta lógica extrae la información necesaria

del paquete encapsulado en el mensaje PACKET_IN para poder crear las reglas que permiten configurar el control *in-band*. Se detectó que no extraía correctamente el puerto de entrada del paquete encapsulado, por lo que las reglas que necesitan de esta información no podían instalarse.

Después de realizar las modificaciones necesarias para obtener correctamente el puerto de entrada, se determinó que era necesario programar una lógica que permitiera analizar los paquetes encapsulados en los mensajes PACKET_IN e instalar las reglas necesarias para implementar el funcionamiento *in-band* de manera autónoma. Para tal propósito, el switch debe ser capaz capaz de detectar el tráfico OpenFlow asociado a otros dispositivos e instalar las reglas necesarias para encaminarlo correctamente.

Por último, y tras lograr el funcionamiento *in-band* del switch BOFUSS, se incorporó la implementción del protocolo Amaru y se modificó la lógica del switch BOFUSS para que sea capaz de reconfigurar su puerto de control haciendo uso de este protocolo.

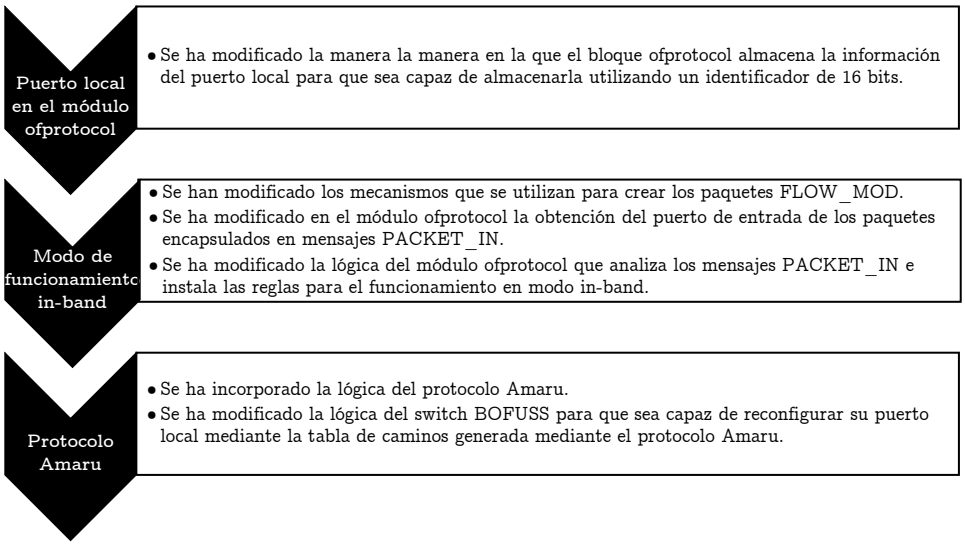


Figura 4.1: Diagrama resumen de las modificaciones implementadas en el switch BOFUSS

4.2 Implementación del puerto local en el bloque ofprotocol

Tras analizar el funcionamiento del bloque `ofprotocol`, se detectó que parte de la lógica implementada para el control *in-band* y la gestión de los paquetes PACKET_IN que envía el módulo `ofdatapath`, no funcionaba correctamente debido a que el bloque `ofprotocol` no incluía el puerto local en sus estructuras de datos.

El bloque `ofprotocol` utiliza los mensajes PORT_DESCRIPTION que el `ofdatapath` envía al controlador para tomar constancia de los puertos del switch y sus características, con el fin de mantener actualizadas las estructuras de datos que emplea. El problema del puerto local viene dado por su identificador, OFPP_LOCAL = `0xfffffffe`, que es de 32 bits, mientras que la estructura del bloque `ofprotocol` que almacena los datos de los puertos tiene un tamaño de 16 bits. Teniendo en cuenta que los puertos se almacenan en función del identificador de número de puerto, el máximo identificador que puede almacenarse es `0xffff`. Este desajuste en el tamaño de los datos provoca que el puerto local no se incluya en la estructura en la que el bloque `ofprotocol` almacena la información de los puertos del switch.

Para solucionar este problema se ha optado por seguir utilizando la misma estructura de 16 bits e implementar el identificador OFPP_LOCAL_AUX = `0xfffe`, que son los 16 bits de menor peso

del identificador OFPP_LOCAL, para almacenar el puerto local en dicha estructura. Esta solución es razonable siempre y cuando el switch cuente con un número de puertos inferior a 65534 (`0xfffe`). Las únicas implicaciones de esta solución son:

1. Cuando el módulo `ofprotocol` obtiene los datos de los puertos del switch de los mensajes PORT_DESCTIPTION, debe comprobar si el identificador del puerto a guardar es el del puerto de control, OFPP_LOCAL = `0xfffffffe`. En caso afirmativo, debe almacenarlo en la posición especificada por el identificador OFPP_LOCAL_AUX.

2. Si se produce un cambio en el puerto local y es necesario actualizar la estructura que contiene los datos de los puertos, se utiliza el identificador OFPP_LOCAL_AUX para localizar el puerto local en dicha estructura.

4.2.1 Modificaciones del código

A continuación, se detallan las funciones que han sufrido cambios para implementar lo mencionado en los párrafos anteriores y se enlaza el código de las modificaciones:

1. Función **port_watcher_local_packet_cb()** del archivo port-watcher.c: Los cambios realizados aparecen en las secciones indicadas por comentarios en el blóque de Código B.1. Estas modificaciones tienen como objetivo lograr que se guarde correctamente el puerto de control. Para alcanzar tal propósito, se trunca el identificador del puerto local a 16 bits, de modo que no exceda el tamaño de la estructura.

2. Función **update_phy_port()** del archivo port-watcher.c: Se comprueba si el puerto buscado es el local para utilizar el identificador de 16 bits (Código B.2).

3. Función **dp_ports_add()** del archivo dp_ports.c: Se ha modificado para que al añadir el puerto local a la lista de puertos del switch active el flag de configuración **OFPPC_NO_FWD**, que evita que se envíen paquetes cuando el switch realice un *FLOOD* (Código B.3), es decir, que mande el paquete por todos sus puertos.

4. Función **dp_ports_output_all()** del archivo dp_ports.c: Se ha modificado para que el switch no envíe el paquete por la interfaz del puerto local cuando haga un *FLOOD*, y así evitar bucles (Código B.4).

4.3 Implementación del modo de control in-band

El modo de control *in-band*, tal y como se ha explicado en la Sección 1.1 del primer capítulo, emplea el plano de datos para transmitir los mensajes de control y administración entre el switch y el controlador. Para que esto sea posible es necesario que el switch sea capaz de encaminar los flujos OpenFlow de distintos dispositivos sin la intervención del controlador.

Una vez se ha solucionado el problema del puerto local, mencionado en el apartado anterior, es necesario modificar el switch para que sea capaz de encaminar los paquetes de otros dispositivos OpenFlow y estos puedan establecer la conexión sin que el controlador deba intervenir en el encaminamiento. Teniendo en cuenta que, al iniciar el switch BOFUSS, el comando `ofprotocol` incorpora la IP del controlador y el puerto OpenFlow para establecer la conexión, los switches ubicados lógicamente entre el controlador y

el nuevo dispositivo deben ser capaces de encaminar los paquetes ARP, enviados por el nuevo dispositivo para conocer la MAC del controlador, y los paquetes TCP de la comunicación OpenFlow.

El switch BOFUSS hace uso de la tabla de flujos para conmutar los paquetes, y en caso de que se produzca un *table-miss*[1], debe descartar los paquetes, tal y como establece la versión 1.3 de la especificación de OpenFlow. Por lo tanto, es necesario instalar en la tabla de flujos unas entradas que permitan encaminar de manera correcta el tráfico mencionado anteriormente.

Antes de proceder a enumerar y explicar las reglas que se instalan, es necesario retomar la Figura 3.1 y recordar cómo es el flujo de los mensajes OpenFlow para comprender el proceso de la implementación de las reglas y sus implicaciones. Se deben tener presentes los siguientes premisas:

1. El socket TCP, mediante el que se establece la conexión OpenFlow entre el controlador y el switch, está asociado a una interfaz asignada como puerto del switch. Este hecho provoca que el switch procese los paquetes de la conexión OpenFlow.

2. Los paquetes OpenFlow que el módulo `ofdatapath` envía al controlador son previamente procesados por el `ofprotocol`.

Teniendo en consideración la segunda premisa, se hace uso de la información extraída de los paquetes encapsulados en los mensajes PACKET_IN para decidir qué reglas deben instalarse en la tabla de flujos para encaminar correctamente el tráfico OpenFlow de los otros dispositivos. Sin embargo, y dado que el switch descarta el flujo cuando se produce un *table-miss*, es necesario instalar unas primeras reglas que indiquen al switch que envíe los paquetes ARP y TCP al controlador, es decir, que genere un PACKET_IN.

En primer lugar, teniendo en cuenta la segunda premisa mencionada anteriormente, es necesario instalar dos reglas que eviten que el switch procese tráfico asociado a su propia conexión OpenFlow con el controlador, dado que este tráfico no debe ser procesado por el `pipeline` del switch BOFUSS. Posteriormente se instalan dos reglas generales, una para el tráfico ARP y otra para el tráfico TCP dirigido al puerto de conexión OpenFlow del controlador. La acción establecida en estas reglas consiste en que el switch envíe los paquetes asociados a dicho tráfico al controlador a través de mensajes PACKET_IN. Por último, y gracias a las reglas anteriores, el `ofprotocol` analiza estos flujos, comprueba si se trata de mensajes ARP que consultan la dirección MAC del controlador, o de mensajes TCP propios del establecimiento de la conexión OpenFlow, e instala en el switch las reglas que permiten establecer la conexión OpenFlow entre el nuevo dispositivo y el controlador. Antes de enumerar las reglas que deben instalarse, es necesario puntualizar que durante el desarrollo del presente TFM se han utilizado los controladores ONOS y RYU para llevar a cabo las pruebas de funcionamiento, tal y como se describe en el Capítulo 5. Debido a que los controladores utilizan aplicaciones de *forwarding* de diferente complejidad, su funcionamiento no es exactamente igual, lo que se traduce en que cada controlador instala diferentes reglas con niveles de prioridad distintos. Las reglas que se instalan para que la implementación funcione correctamente con ambos controladores son las siguientes:

1. Descartar el tráfico recibido en el puerto físico que comparte interfaz con el puerto local, cuya MAC de origen sea la dirección MAC de dicha interfaz.

2. Descartar el tráfico recibido en el puerto físico que comparte interfaz con el puerto local, cuya MAC de destino sea la dirección MAC de dicha interfaz.

3. Enviar el tráfico ARP al controlador.

[1] Quiere decir que el flujo procesado no se asocia a ninguna de las entradas de la tabla de flujos.

4. Enviar al controlador el tráfico TCP cuyo puerto de destino es el puerto en el que se establece la conexión OpenFlow en el controlador.

5. Enviar el tráfico ARP, con destino la MAC del nuevo dispositivo, al puerto por donde se ha recibido el ARP REQUEST, consultando la MAC de la IP del controlador, enviado por el nuevo dispositivo.

6. Enviar el tráfico TCP, con IP destino la del controlador e IP origen la del nuevo dispositivo, por el puerto físico por el que es alcanzable el controlador.

7. Enviar el tráfico TCP, con IP destino la del nuevo dispositivo e IP origen la del controlador, por el puerto físico por el que se ha recibido el tráfico TCP del nuevo dispositivo.

Para que las reglas funcionen correctamente las dos reglas generales instaladas para que envíen el tráfico ARP y TCP al controlador, es decir, las reglas 3 y 4, deben tener menor prioridad que las siguientes reglas que se aprenden dinámicamente. Teniendo esto último en cuenta, las reglas generales se instalan con prioridad `65520`, las reglas que se aprenden dinámicamente se instalan con prioridad `65521` y las reglas DROP con prioridad `65535`, la más alta para evitar que se formen bucles.

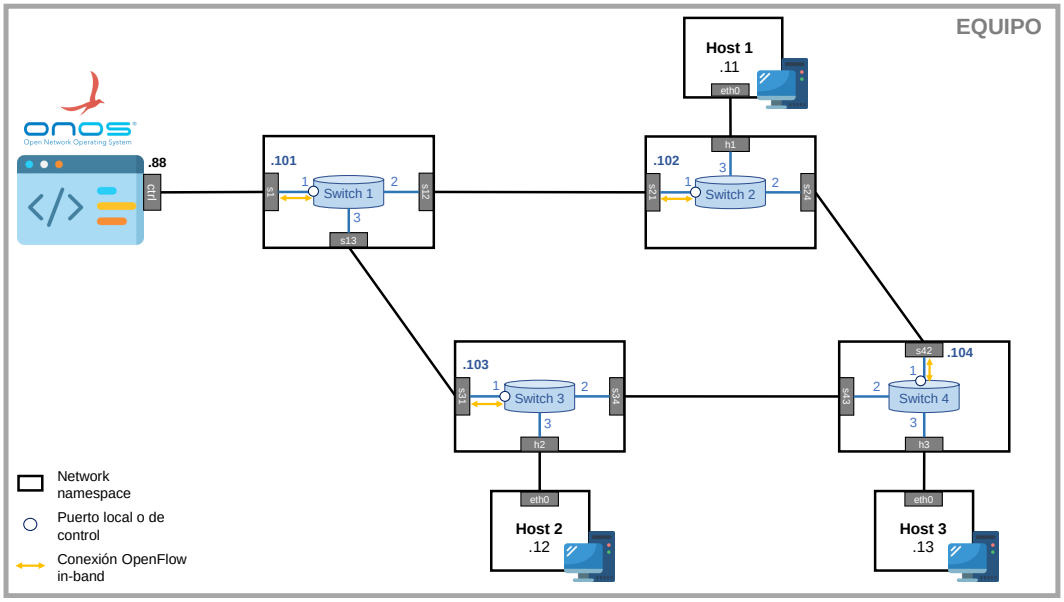


Figura 4.2: Escenario del ejemplo de entradas instaladas en la tabla de flujos de un software-switch

A modo de ejemplo, en la Figura 4.3 se muestra las entradas instaladas en la tabla de flujos del switch 2 de la topología que aparece en la Figura 4.2. Las entradas 5 y 6 representan las reglas generales que hacen que el switch envíe el tráfico ARP y TCP al controlador, aunque en este caso sirve para que el módulo `ofprotocol` procese estos paquetes e instale flujos en caso de que sea necesario. Un ejemplo de flujo TCP para OpenFlow que instala el `ofprotocol` serían las entradas número 3 y 4, que representan las reglas que permiten encaminar el flujo OpenFlow entre el controlador y el dispositivo con IP `10.0.0.104`. El flujo número 3 permite encaminar el tráfico OpenFlow originado en el dispositivo OpenFlow con destino al controlador, mientras que el flujo 4 lo hace en sentido contrario, es decir, encamina el tráfico OpenFlow originado en el controlador con destino el dispositivo con IP `10.0.0.104`.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:02"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="69", byte_cnt="19718", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:02"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="67", byte_cnt="12052", **insts=[]**},

3. match="oxm{**in_port="2", eth_type="0x800", ipv4_src="10.0.0.104", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="76", byte_cnt="18694", **insts=[apply{acts=[out{port="1"}]}]**},

4. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.104", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="79", byte_cnt="11818", **insts=[apply{acts=[out{port="2"}]}]**},

5. match="oxm{**eth_type="0x806"**}", prio="65520", idle_to="0", hard_to="0", pkt_cnt="0", byte_cnt="0", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]**},

6. match="oxm{**eth_type="0x800", ip_proto="6", tcp_dst="6653"**}", prio="65520", idle_to="0", hard_to="0", pkt_cnt="1", byte_cnt="74", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]**},

7. match="oxm{**eth_type="0x806"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="0", byte_cnt="0", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]**},

8. match="oxm{**eth_type="0x8942"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="9", byte_cnt="1260", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]**},

9. match="oxm{**eth_type="0x88cc"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="9", byte_cnt="1260", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]**},

10 match="oxm{**eth_type="0x800"**}", prio="5", idle_to="0", hard_to="0", pkt_cnt="1", byte_cnt="86", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]}**

Figura 4.3: Ejemplo de entradas instaladas en la tabla de flujos de un software-switch

Una vez finalizada la implementación del modo de control *in-band* se debe destacar el detalle de que el puerto local del switch no procesa ni envía mensajes OpenFlow. Si se retoma nuevamente la Figura 3.1, se puede apreciar cómo el flujo de mensajes OpenFlow no transcurre a través del puerto de control del switch BOFUSS, sino que lo hace a través del socket TCP abierto por el módulo OpenFlow a través de la interfaz que permite alcanzar la IP del controlador según la tabla de rutas del sistema. Sin embargo, el puerto local sí que es muy importante porque hace referencia a la interfaz a través de la que se establece esta conexión TCP entre el controlador y el switch, lo que permite identificar el puerto del switch que está asociado a esta interfaz y, por lo tanto, llevar a cabo todas las configuraciones necesarias para la implementación de este modo de control.

4.3.1 Modificaciones del código

Para implementar el modo de control *in-band* se han introducido numerosas modificaciones en el código del switch BOFUSS. En los siguientes apartados se explican los cambios más importantes.

4.3.1.1 Mensajes FLOW_MOD

Teniendo en cuenta que los mensajes FLOW_MOD se utilizan para añadir, eliminar o modificar las entradas de la tabla de flujos, el módulo `ofprotocol` dispone de una lógica pensada para crear este tipo de mensajes y, de este modo, autoinstalar reglas en el switch BOFUSS, es decir, añadir entradas a la tabla de flujos del módulo `ofdatapath` que no sean generadas por el controlador, sino por el propio switch. Por ello, esta funcionalidad es necesaria para poder instalar las reglas que permitan implementar el control *in-band*.

Sin embargo, esta lógica no funciona como es debido. Más concretamente, la estructura de `matches` no se genera correctamente, por lo que el módulo `ofdatapath` es incapaz de extraer los `matches` e implementar adecuadamente las reglas a su tabla de flujos. El objetivo de estas modificaciones es lograr que el módulo `ofprotocol` sea capaz de crear mensajes FLOW_MOD que permitan añadir las

reglas necesarias para el funcionamiento en modo *in-band*, modificarlas o eliminarlas. Las principales modificaciones implementadas son:

- Función **make_flow_mod()** del archivo ofp.c: Se crea correctamente la estructura de `matches` del paquete FLOW_MOD y se rellenan los campos necesarios de los paquetes FLOW_MOD (Código B.5). Para generar correctamente la estructura de `match` se ha implementado la función **create_ofl_match_UAH()** (Código B.6).
- Función **make_add_flow()** del archivo ofp.c: Se ha implementado la creación de un paquete FLOW_MOD de tipo DROP, que instala una regla para descartar el flujo especificado en la estructura de `match` cuando el tamaño de las acciones sea 0, o un FLOW_MOD normal en caso contrario (Código B.7).
- Función **make_add_simple_flow()** del archivo ofp.c: Se ha modificado para que se cree un paquete FLOW_MOD que instale una regla que encamine el tráfico, caracterizado por los campos de `match`, por un puerto concreto. Cuando el puerto de salida es 0, se crea un FLOW_MOD del tipo DROP, es decir, una regla para descartar los paquetes del tráfico caracterizado por los campos de `match` (Código B.8).

4.3.1.2 Configuración del modo in-band

Una vez se ha conseguido que la lógica para generar paquetes FLOW_MOD funcione correctamente, se han realizado las modificaciones pertinentes para configurar adecuadamente el modo de funcionamiento *in-band* del switch BOFUSS. Para ello ha sido necesario modificar la lógica del switch para que sea capaz de instalar las reglas enumeradas en la Sección 4.3. A continuación, se detallan las modificaciones más importantes que se han implementado:

- Se ha modificado la estructura **in_band_data** del archivo de in_band.c: El objetivo de esta modificación es ofrecer acceso a las funciones de *in-band* a lista de puertos del switch que maneja el bloque `ofprotocol` (Código B.13).
- Se ha creado la función **get_of_port_UAH()** en el archivo secchan.c: Su finalidad es obtener el puerto especficado para establecer la conexión OpenFlow con el controlador cuando se inicia el bloque `ofprotocol` (Código B.15).
- Se ha creado la función **install_in_band_rules_UAH()** en el archivo in_band.c: Su propósito es instalar dos reglas DROP que impidan que el switch procese el tráfico de su propia conexión OpenFlow con el controlador, y dos reglas generales para tráfico ARP y TCP a fin de que el bloque `ofdatapath` envíe este tráfico al `ofprotocol`, donde será analizado para poder instalar las reglas que permitan encaminar de manera correcta las conexiones OpenFlow de los nuevos dispositivos. Las reglas generales se instalan con una prioridad CTRL_PRIORITY = `0xfff0`, y las reglas de DROP con la prioridad DROP_PRIORITY = `0xffff`, la más alta para asegurarnos de que este tráfico no se procesa nunca en el `pipeline` del switch. Se han escogido estos valores para las prioridades para evitar que las reglas que puedan ser instaladas por un controlador interfieran en el funcionamineto del modo *in-band* (Código B.16).
- Se ha creado la función **get_pw_local_port_number_UAH()** en el archivo port-watcher.c: El objetivo de esta función es obtener el identificador del puerto físico que comparte interfaz con el puerto de control. Este identificador es imprescindible para poder configurar las reglas que establecen el modo de control *in-band* (Código B.14).

- Función **main()** del archivo secchan.c: Se ha incluido un bloque de código que permita al módulo `ofprotocol` instalar las dos reglas generales para el tráfico ARP y TCP mencionadas anteriormente. Para ello se ha creado la variable global **in_band_rules**. Representa un *flag* que indica si se han instalado las reglas, siendo su principal finalidad evitar que dichas reglas se instalen más de una vez. En el Código B.12 puede comprobarse que, para que se instalen las reglas, deben cumplirse tres condiciones: que se haya establecido la conexión con el módulo `ofdatapath`; que el flag **in_band_rules** esté desactivado; y que se haya obtenido el identificador del puerto físico que comparte interfaz con el puerto local.

- Función **make_packet_out()** del archivo ofp.c: Se ha modificado el método para obtener del tamaño de las acciones porque el tamaño que se obtenía previamente era incorrecto (Código B.9).

- Función **get_ofp_packet_eth_header()** del archivo secchan.c: Se ha sustituido por la función **get_ofp_packet_eth_header_UAH()** que se utiliza para obtener la cabecera Ethernet de un paquete PACKET_IN, así como la estructura **ofl_msg_packet_in** que contiene el mensaje PACKET_IN en formato de bytes de usuario (Código B.10).

- Función **get_ofl_packet_in_()** del archivo secchan.c: Se ha sustituido por la función **get_ofp_packet_in_UAH()** dado que la anterior no funcionaba de manera correcta. La nueva función se utiliza para comprobar si el bloque `ofprotocol` ha recibido a través de la conexión con el `ofdatapath` un mensaje PACKET_IN. En caso de que así sea, se extrae el puerto de entrada del paquete encapsulado en el mensaje PACKET_IN y se hace un `return` del puntero al mensaje PACKET_IN (Código B.11).

- Modificación de la función **in_band_local_packet_cb()** del archivo in-band.c: Esta función es clave dentro de la implementación del modo de control *in-band*. En ella se procesan los mensajes PACKET_IN que el módulo `ofdatapath` envía al controlador. El objetivo es detectar si son mensajes propios del establecimiento de la conexión OpenFlow entre otro dispositivo y el controlador, y en caso de confirmarse, instalar las reglas que posibiliten encaminar correctamente estos mensajes y así permitir que se establezca la conexión. La función **in_band_local_packet_cb()** es invocada cuando el módulo `ofprotocol` recibe datos a través de su conexión local, es decir, el socket mediante el que se comunica con el bloque `ofdatapath`. Cuando esto ocurre, emplea la función **get_ofp_packet_eth_header_UAH()** para intentar extraer el puerto de entrada del paquete que ha generado el PACKET_IN, la cabecera Ethernet y el mensaje PACKET_IN, en caso de haber recibido este tipo de mensaje. Una vez obtenidos estos datos, mediante la función **flow_extract()** se extraen los atributos que caracterizan el flujo del paquete contenido en el mensaje PACKET_IN (codigo B.17). Tras extraer estos datos, se procede a comprobar si el paquete encapsulado es del tipo ARP. En caso afirmativo, se comprueba si el ARP solicita la MAC de la IP del controlador y, si se cumple la condición, procede a instalar la regla para el camino inverso de los ARP, es decir, los mensajes ARP con destino la MAC del nuevo dispositivo son encaminados por el puerto de entrada del mensaje ARP que se está procesando (Código B.18). La prioridad que se utiliza para estas reglas es RULE_PRIORITY = `0xfff1`, que son de menor prioridad que los DROP pero de mayor prioridad que las reglas generales para el tráfico ARP y TCP, previamente instaladas. Si no se trata de un mensaje ARP, se comprueba si es un mensaje TCP con destino el controlador. Si se dan estas dos condiciones, se instalan dos reglas: la primera para el tráfico TCP con IP destino la del controlador e IP origen la del nuevo dispositivo que es encaminado por el puerto físico por donde se alcanza el controlador, y la segunda para el tráfico TCP con destino la del nuevo dispositivo y origen el controlador, que es encaminado por el puerto de entrada del paquete procesado (Código B.19).

4.4 Adaptación de la implementación del protocolo Amaru

La implementación de Amaru que se desarrolla en este proyecto está basada en una implementación existente desarrollada por el grupo GIST - Netserv de la Universidad de Alcalá [31]. Esta implementación base incorpora la lógica necesaria para propagar los mensajes Amaru, generar los caminos disponibles al nodo raíz y almacenarlos, tal y como se detalla en la Sección 2.5.

Las implementaciones incorporadas en este proyecto tienen como objetivo utilizar los caminos generados por Amaru como respaldo en caso de que el enlace asociado al puerto local del switch deje de estar disponible. Por lo tanto, el switch debe detectar cuando el enlace configurado como puerto de control deje de estar operativo para iniciar la configuración de un nuevo puerto local que utilice una de las interfaces asociadas a los caminos disponibles en la tabla de Amaru. Mediante la reconfiguración del puerto de control se recupera la conexión con el controlador y se encaminan correctamente las conexiones OpenFlow de otros dispositivos que estuvieran atravesando el switch en cuestión.

En la Figura 4.4 se representa el diagrama de flujo de las funcionalidades implementadas en esta sección. Primeramente, la lógica de Amaru se inicializa si el switch opera con el modo de control *in-band*. Una vez inicializado el protocolo Amaru, el nodo raíz inicia la exploración, propagando los paquetes utilizados para generar los caminos que posteriormente los nodos almacenarán en sus tablas Amaru. Cuando se detecta la caída del enlace del puerto local, el switch elimina el antiguo puerto local de su lista e inicia la configuración del nuevo puerto de control. Una vez iniciado el proceso, comprueba las entradas de la tabla de Amaru en busca del siguiente camino disponible. Después de localizar el siguiente apto, se consulta el puerto que tiene asociado para que pueda ser configurado como puerto local. Tras identificar dicho puerto, se le asigna la dirección IP que estaba asociada al antiguo puerto local y se envía un mensaje al bloque `ofprotocol` por medio de un mensaje PACKET_IN. Este mensaje PACKET_IN contiene el identificador del antiguo puerto local, el identificador y el nombre de la interfaz del nuevo puerto de control y la IP que tiene asignada. Estos datos son necesarios para que el módulo `ofprotocol` realice los cambios pertinentes en las reglas del modo *in-band*. Cuando el bloque `ofprotocol` recibe este paquete, extrae los datos y con ellos modifica e instala las reglas para el nuevo puerto local. Una vez el módulo `ofprotocol` ha detectado que el puerto local ha cambiado, vuelve a establecer la conexión con el controlador mediante otro socket TCP a través de la interfaz del nuevo puerto local.

Las modificaciones implementadas en las reglas de la tabla de flujos del switch BOFUSS son las siguientes:

1. Se instalan dos reglas DROP para descartar el tráfico recibido en el puerto físico que comparte interfaz con el puerto de control, cuya dirección MAC de origen o destino sea la MAC de la interfaz configurada como puerto de control.

2. Se modifican los flujos TCP con destino la IP del controlador para encaminarlos a través del nuevo puerto local.

3. Se eliminan las entradas DROP asociadas al antiguo puerto local.

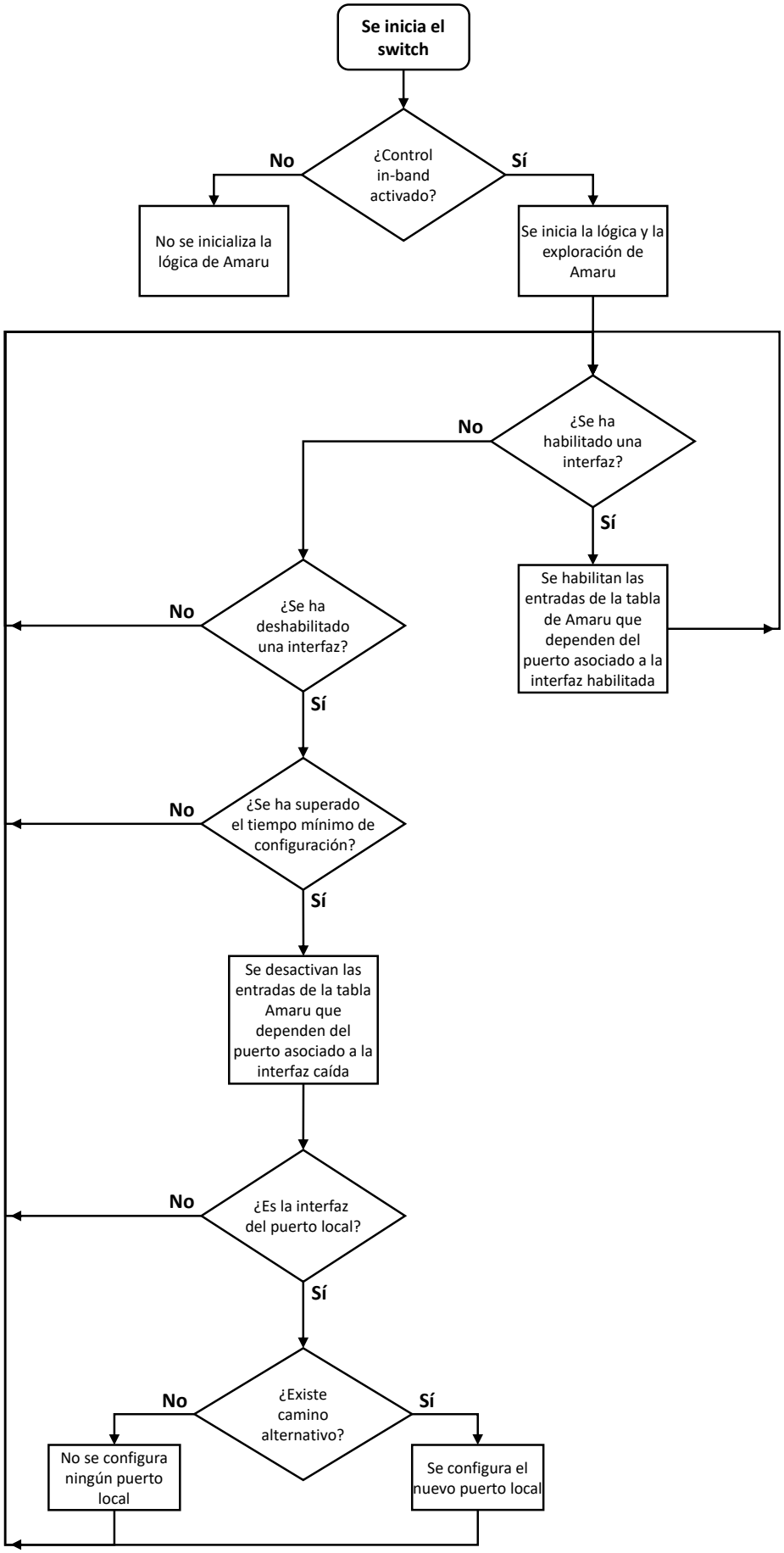


Figura 4.4: Diagrama de flujo de la implementación de Amaru

4.4.1 Modificaciones del código

Antes de implementar la lógica que permita reconfigurar el puerto local del switch, se han efectuado las siguientes modificaciones a la lógica existente de Amaru:

- Estructura **reg_AMAC**: Esta estructura representa una entrada de la tabla de AMACs, o tabla de Amaru. Se ha modificado para que incorpore el flag **active** que indica si la entrada de la tabla se puede utilizar o, por lo contrario, no está disponible debido a que la interfaz de salida está inhabilitada (Código B.20).
- Función **table_AMACS_add_AMAC()**: Esta función se encarga de guardar en una tabla las AMACs recibidas. Se ha modificado para que las AMACs sean almacenadas por orden de llegada, es decir, la primera AMAC disponible en la tabla será la primera que haya recibido el switch, dado que, posiblemente, sea el camino más rápido (Código B.21).
- Función **dp_ports_output_amaru()**: Se encarga de crear y enviar un paquete Amaru por cada uno de los puertos del switch, asignando un sufijo único a la AMAC generada para cada puerto. Se ha modificado el modo de recorrer la lista de puertos para enviar el paquete, y además, se bloquea el envío del paquete por la interfaz del puerto de control (Código B.22).

Una vez realizados estos pequeños cambios, se han implementado las modificaciones en la lógica para que el switch sea capaz de reconfigurar su puerto local haciendo uso de los caminos generados mediante el protocolo Amaru. Los cambios más destacables son los siguientes:

- Se han creado las funciones **disable_invalid_amacs_UAH()** y **enable_valid_amacs_UAH()**: Estas funciones desactivan y activan, respectivamente, el flag **active** de las entradas de la tabla de AMACs en caso de que el puerto al que esté asociada la MAC se deshabilite o habilite (Código B.23).
- Se ha creado la función **remove_local_port_UAH()**: Se encarga de eliminar el puerto local de la lista de puertos del switch y devolver la dirección IP que tenía asignada la interfaz configurada como puerto de control (Código B.24).
- Se ha creado la función **configure_new_local_port_amaru_UAH()**: Se encarga de buscar en la tabla de AMACs una nuevo puerto a través de cuya interfaz se pueda establecer la conexión OpenFlow con el controlador. Para ello, recorre la tabla de Amaru en busca del siguiente camino disponible y obtiene el puerto al que está asociado. Una vez identificado este puerto, se añade como puerto local a la lista de puertos del switch y a la interfaz que tiene asociada se le asigna la IP del antiguo puerto local. Una vez configurado el nuevo puerto de control llama a la función **send_amaru_new_localport_packet_UAH()** para enviar al módulo `ofprotocol` un mensaje que contiene los datos necesarios para configurar y modificar las reglas de la tabla de flujos del `ofdatapath` en función del nuevo puerto local (Código B.25).
- Se ha creado la función **send_amaru_new_localport_packet_UAH()**: Esta función crea el mensaje PACKET_IN que contiene el paquete con todos los datos necesarios para configurar las reglas del nuevo puerto local y lo envía al módulo `ofprotocol`. Para generar el paquete con todos los datos llama a la función **create_amaru_new_localport_packet_UAH** (Código B.26).
- Se ha creado la función **create_amaru_new_localport_packet_UAH**: Crea un paquete que contiene el identificador del nuevo puerto local, el nombre de la interfaz y la IP del y el identificador del antiguo puerto local (Código B.27)

- Función **dp__ports__run()**: Esta función se encarga de verificar periódicamente el estado de los puertos del switch y comprobar si se han recibido datos. Son numerosas las modificaciones que se han implementado en esta función. Para comenzar, si se detecta que se ha habilitado uno de los puertos, se llama a la función **enable__invalid__amacs__UAH()** para rehabilitar las AMACs asociadas a dicho puerto. Además, se ha creado una marca de tiempo, **time__init__local__port**, que se inicializa una vez se ha configurado inicialmente el puerto local. Esta marca temporal se emplea para asegurarse de que el switch finaliza la configuración inicial de sus puertos antes de iniciar la reconfiguración del puerto de control. Si se detecta que se ha deshabilitado un puerto del switch, se actualiza la marca de tiempo y se realizan varias comprobaciones para verificar si se ha deshabilitado el puerto de control. Es necesario hacer énfasis en el proceso mediante el cual se detecta que se ha deshabilitado un puerto porque es crucial para la reconfiguración de un nuevo puerto local. Cuando se detecta que se ha inhabilitado un puerto, el switch realiza varias comprobaciones:

  1. Comprueba si el identificador del puerto coincide con el del puerto local, OFPP_LOCAL = `0xfffffffe`, o si la marca temporal **time__init__local__port** es menor a 5 segundos que es el tiempo de espera que se considera suficiente para que el switch haya terminado de configurar todos sus puertos. Estas comprobaciones se realizan por los siguientes motivos:
     - Si se trata del puerto local no se realizan más acciones sobre este puerto, ya que se trata de un puerto lógico cuya funcionalidad se ha explicado en la Sección 4.2, donde se detalla la implementación del puerto local.
     - Si todavía no se ha superado el tiempo establecido para asegurarse de que el switch configura correctamente sus puertos, no se realiza ninguna operación más porque el switch todavía no ha terminado de inicializarse.
  2. Superadas las condiciones anteriores, se llama a la función **disable__invalid__amacs__UAH()** para deshabilitar las entradas de la tabla de AMACs que dependan del puerto deshabilitado. Seguidamente, se verifica si existe un puerto local ya configurado y si el flag **local__port__ok**, que se emplea para indicar que el puerto local se ha configurado correctamente, está activado. En caso de que se cumplan las dos condiciones, se realizan dos últimas comprobaciones para verificar si la interfaz del puerto inhabilitado coincide con la del puerto local y si el switch no tiene el identificador 1, que es el asignado al switch raíz. Si finalmente satisfacen las dos últimas condiciones, se inicia el proceso de configuración de un nuevo puerto local. Para ello, elimina el antiguo puerto local llamando a la función **remove__local__port__UAH()** y seguidamente invoca a la función **configure__new__local__port__amaru__UAH()**, explicada anteriormente, para configurar el nuevo puerto local (Código B.28).

  Para dar por finalizada la configuración del nuevo puerto local se comprueba si la interfaz asociada al puerto local ha recibido datos correctamente, en tal caso se habilita el flag **local__port__ok** para indicar que el nuevo puerto local está operativo y, de este modo, habilitar la configuración de un nuevo puerto local (Código B.29).

- Se ha creado la función **install__new__localport__rules__UAH()**: Esta función realiza las siguientes modificaciones en la tabla de flujos del `ofdatapath` (Código B.30):
  - Instala las reglas de DROP para descartar el tráfico recibido en el puerto que comparte interfaz con el puerto local, cuya dirección MAC de origen/destino sea la MAC de la interfaz configurada como puerto local.
  - Modifica los flujos TCP con destino la IP del controlador para encaminarlos a través del nuevo puerto de control.

  - Elimina las reglas DROP asociadas al antiguo puerto local.

- Se ha modificado la función **in_band_local_packet_cb()**: Se ha introducido la lógica necesaria para procesar el paquete Amaru enviado por el módulo `ofdatapath`, que incluye los datos necesarios para la modificación de la tabla de flujos acorde al nuevo puerto local, e implementar dichas modificaciones en la tabla de flujos. Para ello, la función comprueba si el paquete incluido en el mensaje PACKET_IN es del tipo Amaru. Si se cumple esta condición se extrae el identificador del nuevo puerto local, el nombre de la interfaz, la IP y el identificador del antiguo puerto local. Con estos datos se llama a la función **install_new_localport_rules_UAH()** para que realice las modificaciones pertinentes en la tabla de flujos del switch (Código B.31).

Capítulo 5

Pruebas y resultados

Locura es hacer lo mismo una y otra vez esperando
obtener resultados diferentes.

Albert Einstein

A lo largo de este capítulo se describen detalladamente los escenarios y las pruebas realizadas para comprobar que las implementaciones que se han llevado a cabo funcionan correctamente. Las pruebas tienen como objetivo validar el funcionamiento de las modificaciones implementadas, detalladas en el capítulo anterior. Las pruebas que se desarrollan en este capítulo no pretenden cuantificar el rendimiento del switch BOFUSS por varios motivos. En primer lugar, no se ha establecido como objetivo del presente TFM. En segundo lugar, el switch BOFUSS está orientado más al prototipado y desarrollo de nuevas funcionalidades, que a obtener un rendimiento elevando en el procesamiento de los paquetes. En último lugar, ya existe un artículo que recoge datos acerca del rendimiento del switch BOFUSS [5].

El capítulo se organiza en tres apartados: en el primero, se explica los elementos básicos que formarán los diferentes entornos de pruebas; el segundo, presenta los diferentes entornos creados y las pruebas que se realizan en cada uno de ellos; el tercero, muestra los resultados obtenidos en las diferentes pruebas.

5.1 Elementos básicos utilizados para generar los escenarios de pruebas

Para verificar que las modificaciones implementadas en el switch BOFUSS funcionan correctamente se han desplegado varios escenarios de pruebas. Algunos de estos escenarios se han construido empleando `network namespaces`, explicados en la Sección 2.8, para asegurarse de que el tráfico de control se transmitía a través de las interfaces correctas, ya que los `network namespaces` cuentan con su propia pila de red, por lo que utilizan su propia tabla de rutas. Para conectar los distintos `network namespaces` y los switches se emplean las interfaces `veth`, explicadas en la Sección 2.9, dando lugar al plano de datos de los diferentes escenarios. En la Figura 5.1 aparecen los elementos básicos mediante los que se van a construir los diferentes escenarios de pruebas.

Para comprobar el correcto funcionamiento de las funcionalidades implementadas se han construido 4 escenarios diferentes. Estos escenarios son lanzados en un único equipo, siendo su gestión más sencilla y rápida ya que es posible crearlos y modificarlos de manera rápida y cómoda. Las direcciones IP utilizadas en los distintos escenarios se han configurado en la red `10.0.0.0/24`. Su construcción se ha automatizado

mediante scripts de bash, facilitando el despliegue y la modificación de los escenarios. En los apartados siguientes se detallan los distintos escenarios así como las pruebas que se han realizado en cada uno de ellos.

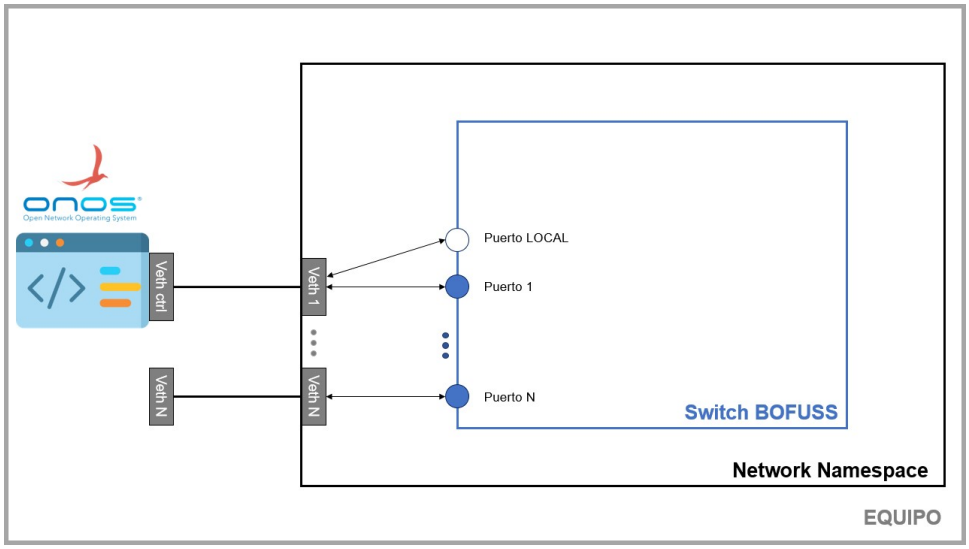


Figura 5.1: Escenario formado por un switch contenido en un network namespace

5.2 Escenarios de pruebas

A lo largo de esta sección se detalla cómo se ha construido cada uno de los 4 escenarios de pruebas que se han utilizado para verificar las modificaciones implementadas en el switch BOFUSS. En los dos primeros escenarios se utilizará también el controlador RYU, además del controlador ONOS . Para llevar a cabo las distintas pruebas se emplean switches que operan en modo *in-band* y en modo *out-of-band*.

- **Switches que utilizan el modo de control *out-of-band***: Estos switches se ejecutan en el `network namespace` raíz, es decir, el `network namespace` por defecto que engloba todo el sistema. Al ejecutar tanto el controlador como los switches en el `network namespace` raíz, la conexión OpenFlow con el controlador se realiza a través de la IP `127.0.0.1` y los mensajes de control OpenFlow se intercambian a través de la interfaz `loopback` del sistema.
- **Switches que utilizan el modo de control *in-band***: Cada uno de estos switches debe levantarse dentro de un `network namespace` diferente. Además, una de las interfaces de cada `network namespace` debe configurarse con una IP que permita establecer la conexión con el controlador y funcionar como puerto de control del switch. Para los switches que operan en este modo de control se han asignado IPs a partir de la dirección `10.0.0.100/24`. Por último, se debe mencionar que para realizar las diferentes pruebas, se han configurado estos switches para que en sus tablas de Amaru almacenen como máximo 8 entradas válidas, puesto que se considera un número suficientemente alto para los requisitos de las pruebas.

En algunos escenarios se han desplegado *hosts*, también conocidos como nodos finales, para simular escenarios más reales y poder comprobar que, aunque se reconfiguren los switches de la red, los *hosts* mantienen la capacidad de comunicarse entre sí. Para crear los *hosts* de los escenarios se levantan tantos `network namespaces` diferentes como *hosts* se requieran y a cada `network namespace` se le configura una interfaz con una IP a partir de la dirección `10.0.0.10/24`.

Los dos primeros escenarios están más enfocados en comprobar que el switch no ha perdido las funcionalidades que ofrecía antes de incorporar las distintas modificaciones y en verificar que las topologías con switches operando en distintos modos de control son igualmente estables y funcionales. Por otra parte, los escenarios 3 y 4 se centran en verificar que el switch es capaz de recuperarse ante la caída de su puerto de control utilizando la información generada por el protocolo Amaru para reconfigurar uno nuevo. Por este motivo, en estos escenarios se han previsto situaciones de caídas de enlaces que permitan comprobar que la reconfiguración del puerto local se realiza de manera correcta, y, al mismo tiempo, que se ofrece mayor robustez y seguridad a la red de control *in-band*, tal como se especifica en la Sección 3.1, puesto que permite seguir encaminando de manera correcta las conexiones OpenFlow de los otros dispositivos de la red que pudieran atravesar el switch que ha reconfigurado su puerto de control.

Además, los distintos escenarios deben cumplir los siguientes requisitos:

- El switch raíz (*root*), es decir, el dispositivo conectado directamente con el controlador, debe configurarse con el identificador 1 para que sea este nodo el que inicie la exploración de Amaru.
- Durante las pruebas, el nodo raíz no puede reconfigurar su puerto de control, ya que tendrá una sola interfaz conectada al controlador.
- Las interfaces de los `network namespaces` que estén o puedan configurarse como puerto de control deben tener desactivado el ***TCP checksum offloading*** [37] **de transmisión** para que se pueda establecer la conexión OpenFlow con el controlador. Esto se debe a que al utilizar enlaces virtuales, como los `veth`, si se delega el cálculo del *checksum* de la cabecera TCP a la tarjeta de red física no se calcula correctamente porque en los diferentes escenarios no se utiliza ninguna y, por lo tanto, los paquetes se descartarán cuando se intente establecer la conexión con el controlador por tener un *checksum* incorrecto. Para deshabilitarlo se utiliza la herramienta `ethtool` [38]. El siguiente bloque de código muestra un ejemplo de uso:

Código 5.1: Comando para desactivar el TCP checksum offloading de un dispositivo veth

```
1 ethtool -K [nombre_interfaz] tx off
```

- El controlador RYU debe ejecutar la aplicación `SimpleSwitch13`. Se trata de una aplicación de controlador que utiliza la versión 1.3 de OpenFlow e implementa un switch estándar de capa 2 para llevar a cabo las diferentes pruebas. Esta aplicación se encuentra en la ruta `ryu/app/` del directorio del paquete de RYU.
- El controlador ONOS debe utilizar la aplicación `Reactive Forwarding`. Esta aplicación permite implementar un switch estándar con la diferencia de que computa las rutas de manera centralizada en el controlador, evitando que se generen bucles en la red.
- Los escenarios que desplieguen switches con el modo de control *in-band* deberán tener configuradas inicialmente, en cada `network namespace`, las interfaces que los switches utilizarán para establecer la conexión OpenFlow en el momento de iniciarlos.

5.2.1 Configuración de los controladores utilizados en los escenarios

Para llevar a cabo las pruebas se utiliza la versión 2.2 del controlador ONOS y la versión 4.34 de RYU. Ambos controladores se han instalado seguiendo las instrucciones recogidas en los manuales A.2 y A.3, respectivamente. Para iniciar la aplicación `Simple Switch 13` de RYU, considerando que se inicia desde el directorio raíz del paquete RYU, se ejecuta el siguiente comando:

Código 5.2: Comando para iniciar la aplicación SimpleSwitch13 de RYU

```
1 ryu-manager --verbose /ryu/app/simple_switch_13.py
```

Antes de iniciar el controlador ONOS, es necesario modificar su configuración para que permita que los *software-switches* tengan instaladas reglas ajenas al controlador, dado que, por defecto, ONOS elimina las reglas que no haya instalado el propio controlador. Para ello, es necesario asignar el valor ***true*** a la constante **`ALLOW_EXTRANEOUS_RULES_DEFAULT`** del archivo `OsgiPropertyConstants.java`, ubicado en la ruta `/onos/core/net/src/main/java/org/onosproject/net/` del paquete de ONOS. Para iniciar el controlador ONOS se ejecuta el siguiente comando desde el directorio raíz del paquete de ONOS:

Código 5.3: Comando para iniciar ONOS

```
1 bazel run onos-local
```

Una vez se ha iniciado ONOS, se activa la aplicación `Reactive Forwarding` en la pestaña de aplicaciones de la interfaz gráfica de ONOS, a la cual se accede mediante la URL `localhost:8181/onos/ui` y las credenciales `onos\rocks`.

5.2.2 Escenario 1: Topología formada por tres switches funcionando en modo de control out-of-band

Este escenario se ha construido con el objetivo de comprobar que las nuevas implementaciones incorporadas al *software-switch* no han alterado el funcionamiento que el switch BOFUSS desempeñaba previamente. Por lo tanto, su principal cometido es probar que el switch BOFUSS es capaz de operar correctamente en el modo de control *out-of-band*.

En la Figura 5.2 aparece representado este escenario. Está formado por 3 switches, dos *hosts* y el controlador. Aunque en la figura aparezca únicamente el logo de ONOS, en este escenario también se utiliza el controlador RYU. Los 3 switches tienen activado el modo de control *out-of-band*, es decir, no utilizan los enlaces existentes entre los switches para establecer la conexión OpenFlow con el controlador, sino que utilizan una red dedicada, en este caso está definida por la interfaz de `loopback`.

Para este escenario se han creado 2 *scripts* de *bash*, uno para crear los `network namespaces` y las interfaces `veth` (*script* C.1) y otro para crear los switches del escenario (*script* C.7). En este último se observa que al crear los switches se especifica que el modo de control es *out-of-band*, ya que por defecto el switch BOFUSS funciona en modo *in-band*. Además, según lo que se ha explicado anteriormente sobre los switches que funcionan en modo de control *out-of-band* en los escenarios de pruebas, la conexión con el controlador se establece a través de la dirección IP `127.0.0.1`, asociada a la interfaz de `loopback` del sistema.

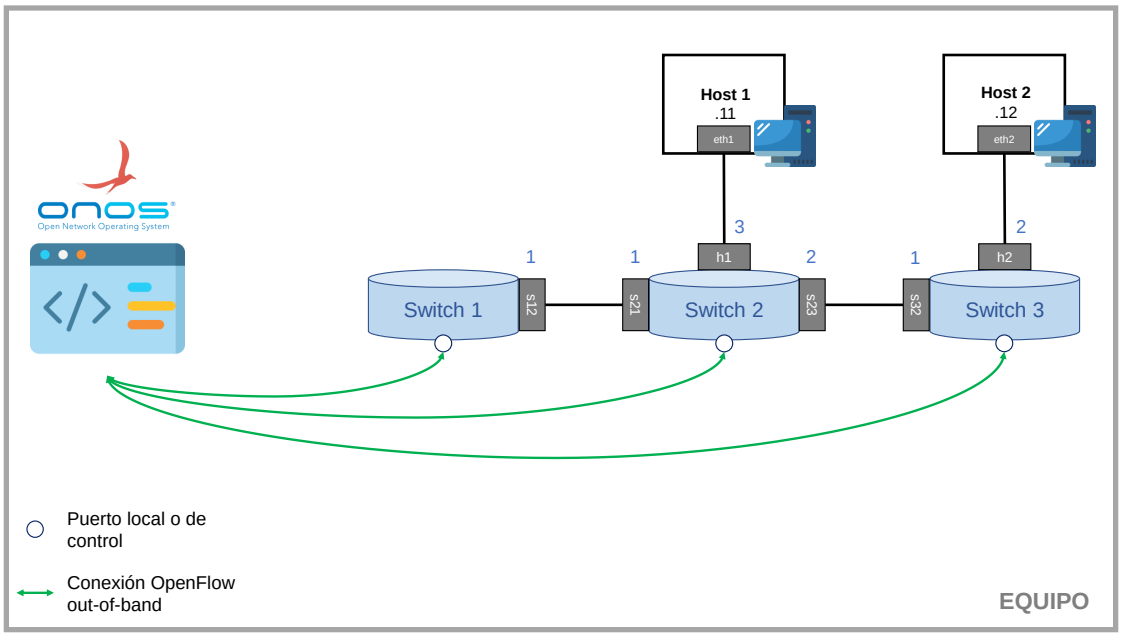


Figura 5.2: Topología escenario 1: 3 switches funcionando en modo de control out-of-band

5.2.3 Escenario 2: Topología formada por dos switches funcionando en modo de control in-band y uno en modo out-of-band

El segundo escenario se ha creado con el propósito de verificar que el switch BOFUSS se mantiene estable en una topología en la que coexisten switches que operan en el modo de control *in-band* con otros que lo hacen en el modo de control *out-of-band*.

En la Figura 5.3 se muestra la topología del escenario. Está formada por 3 switches, 2 switches que operan en modo *in-band* y uno en modo *out-of-band*, 2 *hosts* y el controlador. En este segundo escenario se vuelve a emplear ONOS y RYU como controladores. Para crear el escenario e iniciar los switches se han utilizado los *scripts* C.2 y C.8, respectivamente.

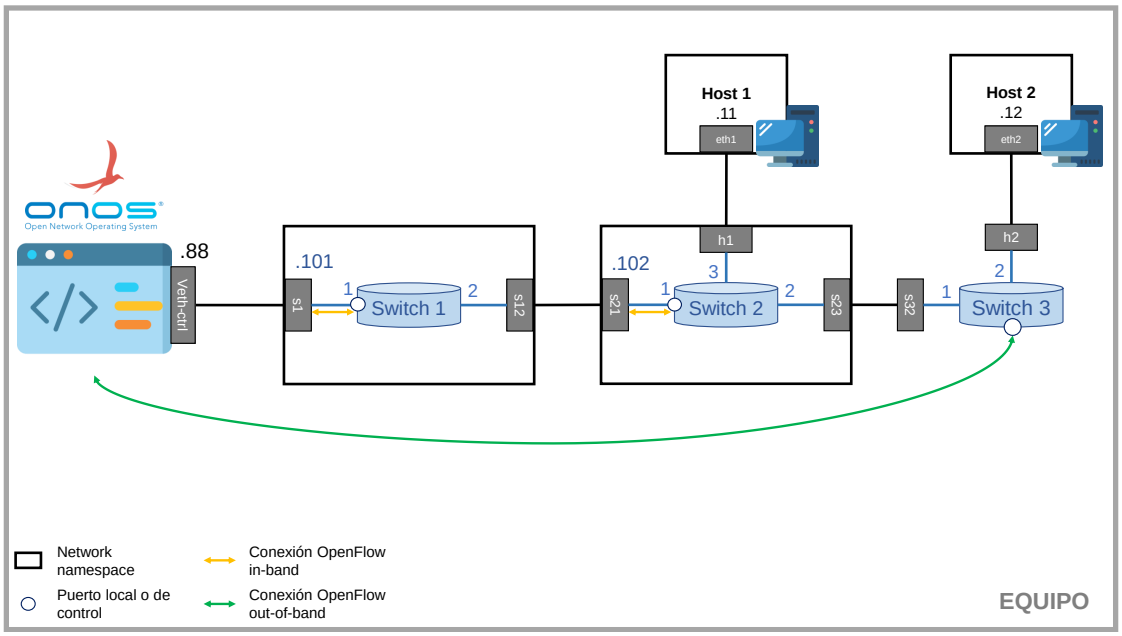


Figura 5.3: Topología escenario 2: 2 switches funcionando en modo de control in-band y 1 out-of-band

5.2.4 Escenario 3: Topología formada por 4 switches funcionando en modo de control in-band

El tercer escenario se ha creado con el objetivo de comprobar el funcionamiento del modo *in-band*, así como la reconfiguración del puerto de control mediante el protocolo Amaru. Para ello, se ha creado una topología formada por 4 switches que operan con el modo de control *in-band* y 3 *hosts*. Los *hosts* en este escenario son interesantes ya que verificar la comunicación entre ellos permite, en primer lugar, confirmar que los elementos de la red se han inicializado correctamente y, en segundo lugar, verificar la robustez de la red al comprobar si se ha recuperado correctamente ante la caída de enlaces. En este entorno únicamente se utiliza ONOS como controlador y su aplicación `Reactive Forwarding` porque, debido al interconexionado de los nodos de la topología, es necesario que se evite la generación de bucles en la red.

Se han creado dos versiones de esta topología modificando el nivel de conexión entre los switches. En la primera, representada en la Figura 5.4, se utiliza un nivel de conexión 2, es decir, cada switch está conectado con otros dos. Para crear el escenario se ha utilizado el *script* C.3 y para iniciar los switches se ha utilizado el *script* C.9. En la segunda versión, representada en la Figura 5.5, se utiliza un nivel de conexión 3, propiciando que Amaru genere más caminos alternativos hasta el nodo raíz. Los nuevos enlaces añadidos permiten verificar el comportamiento de la red al existir la posibilidad de crearse bucles. Para levantar el escenario se ha utilizado el *script* C.4 y para iniciar los switches se ha utilizado el *script* C.10.

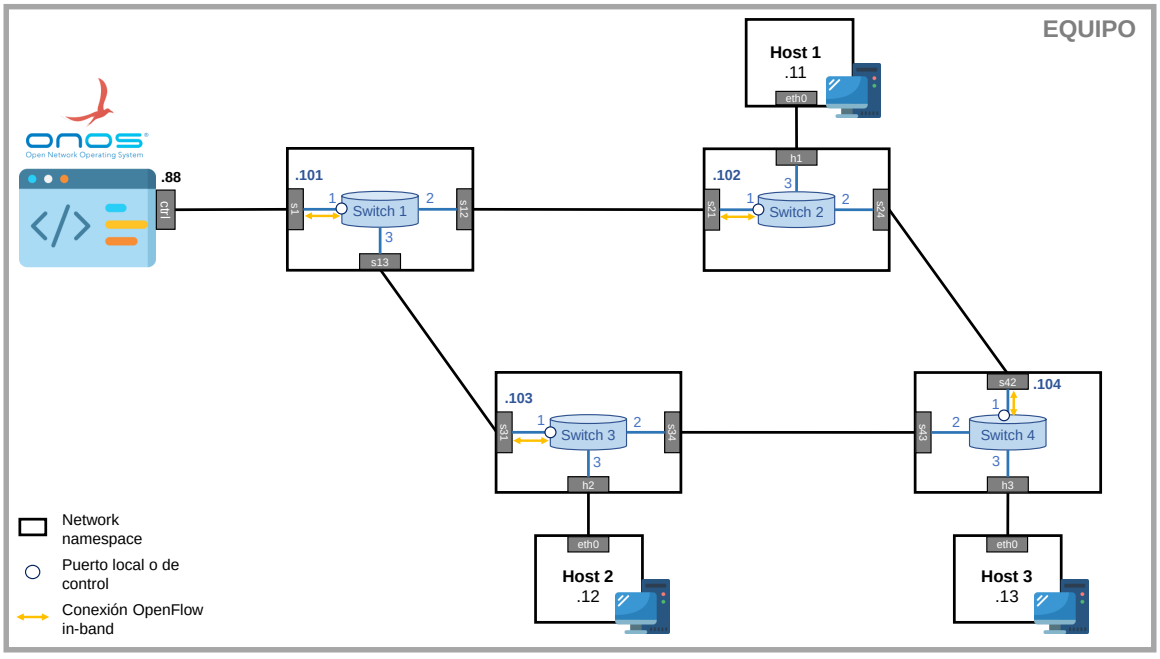


Figura 5.4: Topología escenario 3 (N=2): Nivel de conectividad 2 entre switches

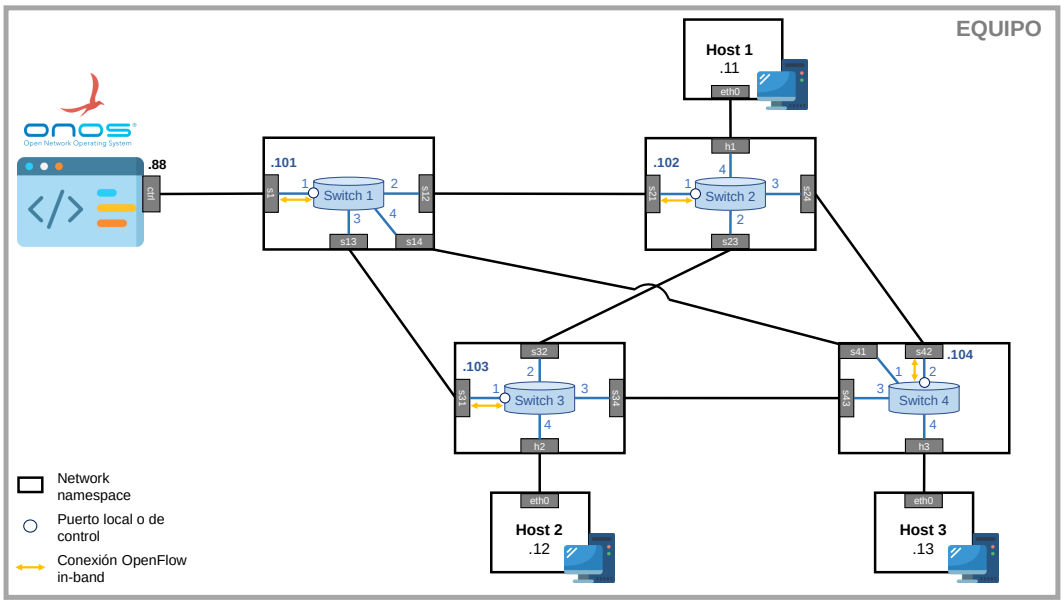


Figura 5.5: Topología escenario 3 (N=3): Nivel de conectividad 3 entre switches

5.2.5 Escenario 4: Topología en forma de triple rombo funcionando en modo de control in-band

El cuarto escenario se ha diseñado para crear una topología más compleja con el objetivo de comprobar que la implementación que permite reconfigurar el puerto local, haciendo uso de los caminos alternativos generados por el protocolo Amaru, funciona correctamente cuando el puerto de control de un switch BOFUSS se reconfigura varias veces.

La topología de este escenario se muestra en la Figura 5.6. Está formada por el controlador ONOS y 6 switches funcionando con el modo de control *in-band*, conectados entre sí de tal modo que existan múltiples caminos hasta el nodo raíz y así poder comprobar que la reconfiguración se hace de manera correcta y no se forman bucles. Para generar el entorno se ha creado el *script* C.6 y para iniciar los switches el *script* C.12.

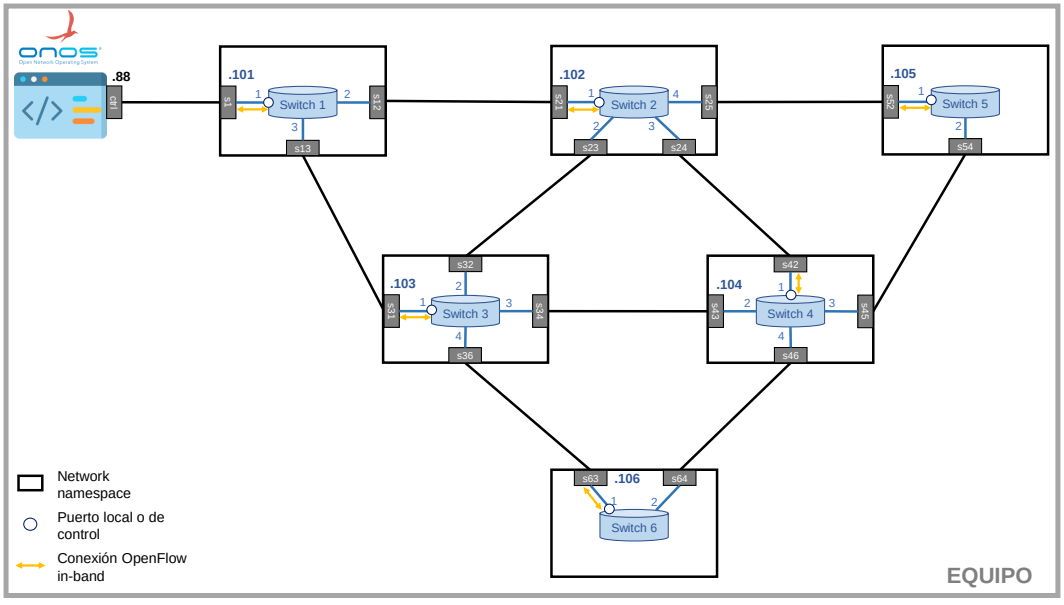


Figura 5.6: Topología del escenario 4

5.3 Resultados experimentales

A lo largo de esta sección se describen las pruebas realizadas en cada uno de los escenarios y se analizan los resultados obtenidos. Tal y como se ha ido mencionando a lo largo de este capítulo, el conjunto de pruebas realizadas tiene tres objetivos principales:

- Comprobar que no se ha alterado el funcionamiento que desempeñaba el switch antes de implementar las modificaciones.
- Comprobar que el switch opera correctamente empleando el modo de control *in-band* que se ha implementado.
- Comprobar que el switch es capaz de reconfigurar su puerto de control en base a los caminos generados mediante el protocolo Amaru.

5.3.1 Pruebas realizadas en escenario 1: Topología formada por tres switches en modo de control out-of-band

El objetivo de las pruebas llevadas a cabo en este escenario (Sección 5.2.2) es comprobar que no se ha alterado el funcionamiento que el switch desempeñaba antes de incorporar las nuevas implementaciones. Para llevar a cabo los tests, se ha creado el escenario con cada uno de los controladores RYU y ONOS. Las pruebas realizadas son:

- Comprobar mediante el analizador de paquetes `Wireshark`[1] que los switches establecen la conexión OpenFlow través de la interfaz de `loopback`.
- Comprobar que los *hosts* pueden comunicarse entre sí utilizando la herramienta `ping` [39].

5.3.1.1 Resultados utilizando el controlador ONOS

A lo largo de esta sección se describen los resultados obtenidos utilizando ONOS como controlador en la topología del escenario 1 donde los switches utilizan el modo de control *in-band*. Una vez iniciados los dispositivos del escenario se ha configurado `Wireshark` para que analice la interfaz de `loopback`. A continuación, se han lanzado 5 `pings`, en ambos sentidos, entre el *host* 1 (IP `10.0.0.11`) conectado al switch 2, y el *host* 2 (IP `10.0.0.12`) conectado al switch 3. Los resultados obtenidos por línea de comandos se muestran en la Figura 5.7, donde se puede observar que existe comunicación en ambos sentidos.

[1] https://www.wireshark.org/

Se realizan los pings desde el **host 1**:
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
64 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=34.6 ms
64 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=2.08 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=1.49 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=1.61 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=1.45 ms
--- 10.0.0.12 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 1.457/8.268/34.692/13.213 ms

Se realizan los pings desde el **host 2**:
PING 10.0.0.11 (10.0.0.11) 56(84) bytes of data.
64 bytes from 10.0.0.11: icmp_seq=1 ttl=64 time=4.03 ms
64 bytes from 10.0.0.11: icmp_seq=2 ttl=64 time=1.73 ms
64 bytes from 10.0.0.11: icmp_seq=3 ttl=64 time=1.61 ms
64 bytes from 10.0.0.11: icmp_seq=4 ttl=64 time=1.95 ms
64 bytes from 10.0.0.11: icmp_seq=5 ttl=64 time=1.56 ms
--- 10.0.0.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 1.563/2.179/4.031/0.937 ms

Figura 5.7: Escenario 1 (ONOS): Resultados obtenidos mediante la herramienta ping

Una vez comprobado que existe comunicación entre ambos *hosts*, se ha analizado la captura de `Wireshark` aplicando el filtro `openflow_v4` para visualizar únicamente el tráfico OpenFlow 1.3. De este modo se ha podido verificar que los mensajes de las conexiones OpenFlow de los switches se transmiten a través de la interfaz de `loopback`. Las Figuras 5.8 y 5.9 muestran ejemplos de mensajes PACKET_IN generados por los switches. En la primera se muestra el PACKET_IN generado por el switch 2 cuando el *host* 1 realiza un ARP REQUEST, para poder conocer la MAC del *host* 2 y poder enviar el `ping`. En la segunda muestra el PACKET_IN generado por el switch 2 cuando el *host* 1 envía el primer `ping`.

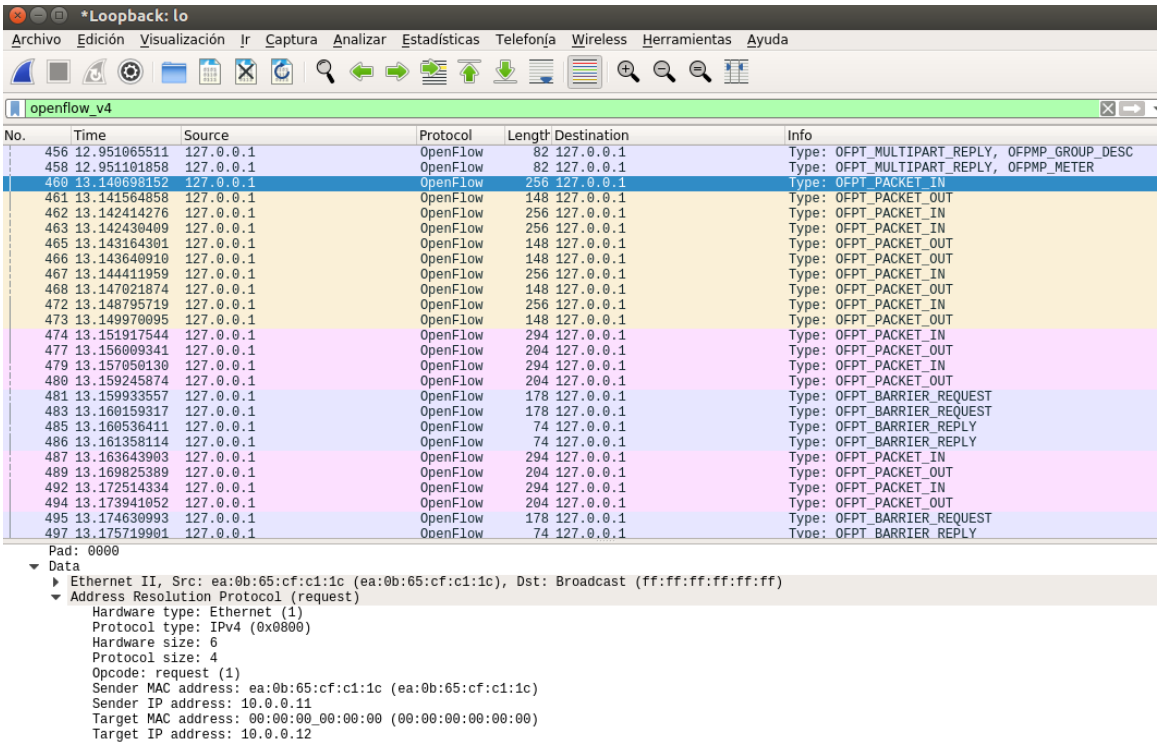


Figura 5.8: Escenario 1 (ONOS): Mensajes PACKET_IN generados por el ARP REQUEST

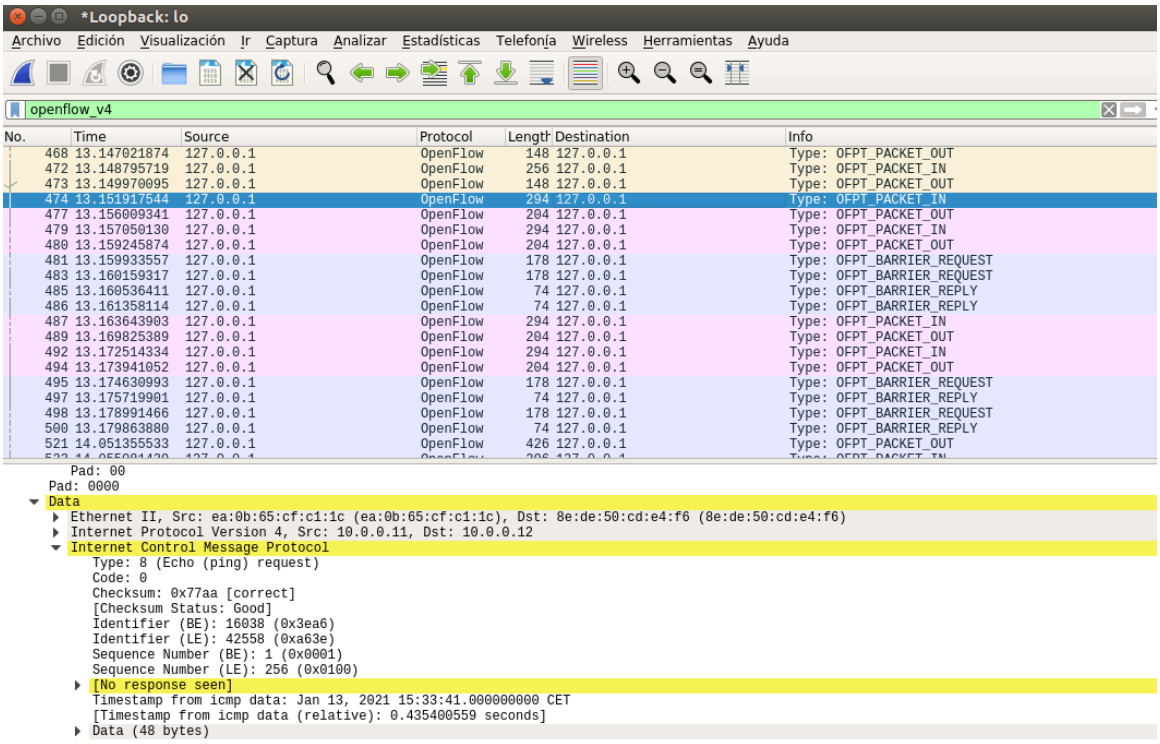


Figura 5.9: Escenario 1 (ONOS): Mensajes PACKET_IN generados por los pings

Tras haber lanzado los `pings`, en la interfaz gráfica de ONOS se puede apreciar la topología completa del escenario (Figura 5.10), donde pueden observarse los 3 switches y los 2 hosts que forman parte de la topología, tal y como se ha diseñado el escenario.

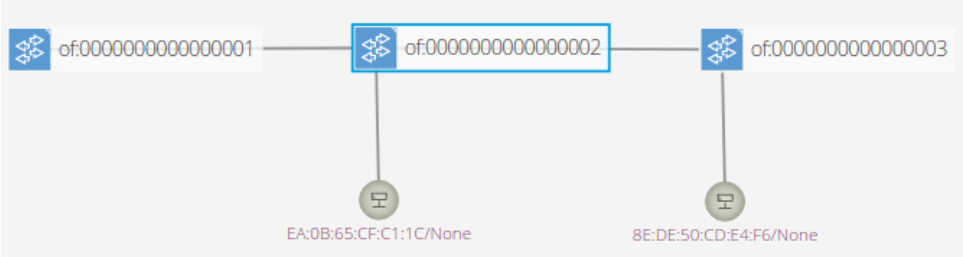


Figura 5.10: Escenario 1 (ONOS): Topología del escenario obtenida en la interfaz gráfica de ONOS

Además, se han obtenido las reglas de la tabla de flujos de los switches, antes y después de realizar los `pings`. La Figura 5.11 muestra el contenido inicial de la tabla de flujos del switch 2, siendo iguales las tablas de flujos del resto de switches. En esta imagen se puede comprobar que ONOS instala 4 reglas que se aplican al tráfico ARP, LLDP[2], BDDP[3] e IP, respectivamente. En la Figura 5.12 se muestra las reglas que tiene instaladas el switch 2 tras haberse realizado los `pings` en amabas direcciones. La única diferencia con las reglas iniciales es que ONOS ha instalado una regla para cada sentido para el tráfico entre los dos *hosts*. En cuanto a la tabla de flujos del switch 3, esta sería igual que la del switch 2. Por otra parte, la tabla del switch 1 no se habría visto afectada por los `pings`, ya que no interviene en la comunicación entre los dos *hosts*.

[2]https://en.wikipedia.org/wiki/Link_Layer_Discovery_Protocol
[3]https://www.openflow.es/2018/05/28/bbdp.html

1. match="oxm{**eth_type="0x806"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="2", byte_cnt="120", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]**},

2. match="oxm{**eth_type="0x88cc"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="60", byte_cnt="8400", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]**},

3. match="oxm{**eth_type="0x8942"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="60", byte_cnt="8400", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]**},

4. match="oxm{**eth_type="0x800"**}", prio="5", idle_to="0", hard_to="0", pkt_cnt="3", byte_cnt="294", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]**}

Figura 5.11: Escenario 1 (ONOS): Entradas de la tabla de flujos del switch 2 al contectarse con el controlador

1. match="oxm{**eth_type="0x806"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="2", byte_cnt="120", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}**},

2. match="oxm{**eth_type="0x88cc"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="82", byte_cnt="11480", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]**},

3. match="oxm{**eth_type="0x8942"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="82", byte_cnt="11480", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]**},

4. match="oxm{**in_port="3", eth_dst="8e:de:50:cd:e4:f6", eth_src="ea:0b:65:cf:c1:1c"**}",prio="10", idle_to="0", hard_to="0", pkt_cnt="9", byte_cnt="882", **insts=[apply{acts=[out{port="2"}]}]**},

5. match="oxm{**in_port="2", eth_dst="ea:0b:65:cf:c1:1c", eth_src="8e:de:50:cd:e4:f6"**}", prio="10", idle_to="0", hard_to="0", pkt_cnt="9", byte_cnt="882", **insts=[apply{acts=[out{port="3"}]}]**},

6. match="oxm{**eth_type="0x800"**}", prio="5", idle_to="0", hard_to="0", pkt_cnt="5", byte_cnt="490", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]**}

Figura 5.12: Escenario 1 (ONOS): Entradas de la tabla de flujos del switch 2 tras lanzar los pings entre los 2 hosts

Mediante las pruebas realizadas en este escenario, se ha verificado que el switch BOFUSS mantiene la capacidad de funcionar del mismo modo que lo hacía antes de implementar las nuevas funcionalidades.

5.3.1.2 Resultados utilizando el controlador RYU

En este apartado se describen los resultados obtenidos utilizando el controlador RYU. En primer lugar se ha iniciado el escenario utilizando la aplicación `Simple Switch 13` de RYU (5.2). Seguidamente, se han lanzado 5 `pings` en cada sentido entre el *host* 1 (IP `10.0.0.11`) y el *host* 2 (IP `10.0.0.12`), obteniendo los resultados que se muestran en la Figura 5.13. De estos resultados se puede concluir que existe comunicación entre ambos *hosts*.

Se realizan los pings desde el **host 1**:
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
64 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=28.1 ms
64 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=1.53 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=1.16 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=1.78 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=1.79 ms
--- 10.0.0.12 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 1.165/6.877/28.113/10.620 ms

Se realizan los pings desde el **host 2**:
PING 10.0.0.11 (10.0.0.11) 56(84) bytes of data.
64 bytes from 10.0.0.11: icmp_seq=1 ttl=64 time=1.68 ms
64 bytes from 10.0.0.11: icmp_seq=2 ttl=64 time=1.55 ms
64 bytes from 10.0.0.11: icmp_seq=3 ttl=64 time=1.55 ms
64 bytes from 10.0.0.11: icmp_seq=4 ttl=64 time=1.83 ms
64 bytes from 10.0.0.11: icmp_seq=5 ttl=64 time=1.54 ms
--- 10.0.0.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 1.541/1.633/1.837/0.116 ms

Figura 5.13: Escenario 1 (RYU): Resultados obtenidos mediante la herramienta ping

En primer lugar, se ha comprobado mediante la herramienta `Wireshark` que los paquetes OpenFlow se transmiten a través de la interfaz de `loopback`. Sin embargo, no se han añadido capturas porque son muy similares a las que se han obtenido con el controlador ONOS. A continuación, se han analizado las reglas que los switches tienen instaladas antes y después de realizar los *pings*. La Figura 5.14 refleja ambos momentos en el switch 2. En la parte superior de la figura se puede apreciar que el switch RYU únicamente instala una regla antes de lanzar los *pings*. Esta regla provoca que el switch genere un PACKET_IN por cualquier tipo de tráfico que reciba. En la parte inferior se observa que el controlador instala un regla en cada sentido para el tráfico entre los dos hosts. La tabla de flujos del switch 3 no se muestra pero en esencia es igual que la del switch 2, mientras que la tabla de flujos del switch 1 no se ha visto afectada por los *pings* porque no interviene en la comunicación entre ambos *hosts*.

1. match="oxm{**all match**}", prio="0", idle_to="0", hard_to="0", pkt_cnt="0", byte_cnt="0", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]**}|

1. match="oxm{**in_port="2", eth_dst="56:b4:77:49:7e:f4", eth_src="ca:68:e8:af:48:ce"**}",prio="1", idle_to="0", hard_to="0", pkt_cnt="11", byte_cnt="1040", **insts=[apply{acts=[out{port="3"}]}]**},

2. match="oxm{**in_port="3", eth_dst="ca:68:e8:af:48:ce", eth_src="56:b4:77:49:7e:f4"**}",prio="1", idle_to="0", hard_to="0", pkt_cnt="10", byte_cnt="980", **insts=[apply{acts=[out{port="2"}]}]**},

3. match="oxm{**all match**}",prio="0", idle_to="0", hard_to="0", pkt_cnt="3", byte_cnt="218", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]}]}]**}

Figura 5.14: Escenario 1 (RYU): Entradas de la tabla de flujos del switch 2 antes y después de realizar los pings entre los 2 hosts

5.3.2 Pruebas realizadas en el escenario 2: Topología formada por dos switches en modo de control in-band y uno en modo out-of-band

En este segundo escenario (Figura 5.3) el objetivo es comprobar que la red funciona correctamente cuando está formada tanto por switches que operan en el modo de control *in-band*, como por switches que tienen activado el modo de control *out-of-band*.

Del mismo modo que en el escenario anterior, se han utilizado los controladores ONOS y RYU para realizar las comprobaciones programadas. Las pruebas que se han realizado han sido las mismas que en el escenario anterior. Sin embargo, como la topología tiene desplegados switches que operan con distintos modos de control, se ha comprobado si los paquetes OpenFlow de los switches que funcionan en modo de control *in-band* se transmiten mediante los enlaces correctos y no mediante la interfaz de `loopback`, como debe ocurrir en el caso de los switches que utilicen el modo de control *out-of-band*.

5.3.2.1 Resultados utilizando ONOS

Tras iniciar el escenario 2 utilizando el controlador ONOS, se han obtenido varias capturas interesantes mediante `Wireshark`. La Figura 5.15 muestra un conjunto de mensajes capturados en la interfaz `veth-ctrl`, que representa la interfaz a través de la que ONOS recibe los paquetes provenientes del `network namespace` en el que está ubicado el switch 1. En la captura se pueden observar el tráfico ARP y TCP mediante el que el switch 2 establece la conexión con el controlador, además de los paquetes OpenFlow del switch 1. De esta manera, se verifica que el switch 1 y el switch 2 están funcionando en modo de control *in-band* dado que los paquetes OpenFlow del switch 2 son encaminados por el switch 1.

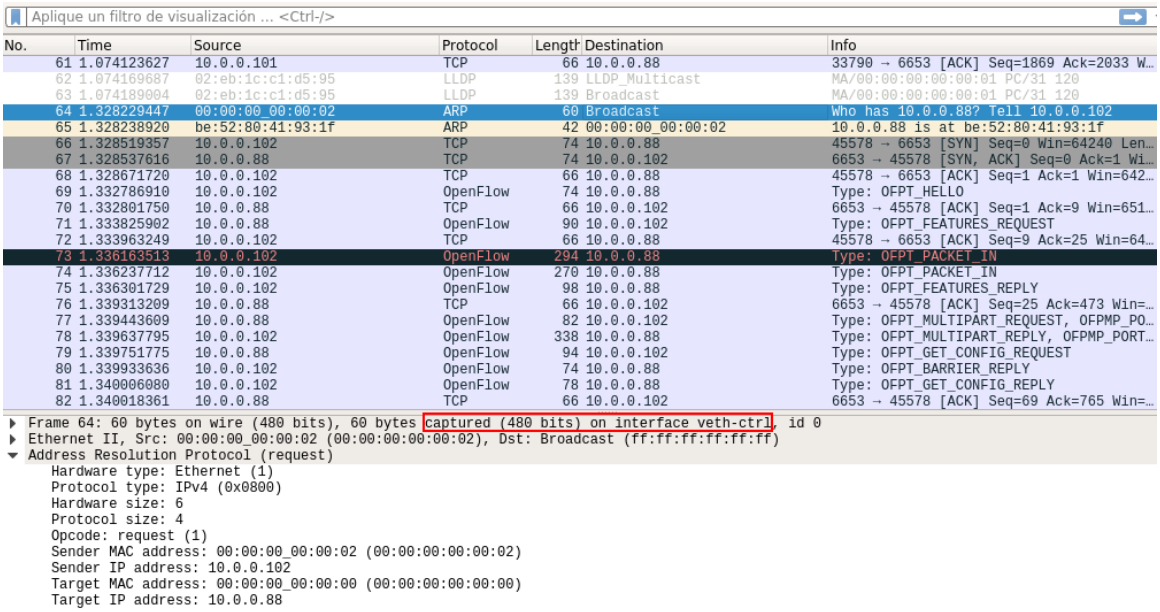


Figura 5.15: Escenario 2 (ONOS): Mensajes de la conexión del switch 2 con el controlador en la interfaz veth-ctrl

Además, en la Figura 5.16 se puede comprobar que el controlador ONOS tiene constancia de que la interfaz `veth-s21`, que se corresponde con el enlace que comunica el switch 2 con el switch 1, está configurada como puerto de control indicando que a través de ella deben transmitirse los paquetes OpenFlow del switch 2.

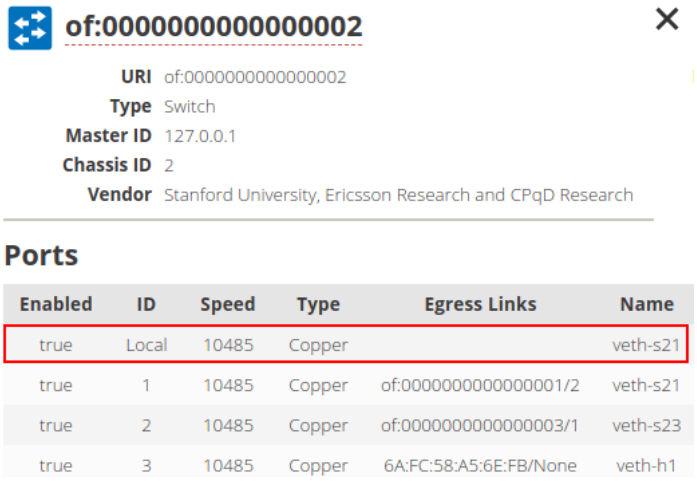


Figura 5.16: Escenario 2 (ONOS): Puerto local del switch 2 en la lista de puertos recibida por el controlador

Tras haber obtenido las capturas mencionadas anteriormente, se ha analizado la tabla de flujos del switch 1, cuyo contenido se recoge en la Figura 5.17. En esta imagen destacan las 7 primeras entradas que son instaladas por la implementación del modo de control *in-band*, ya que las 4 siguientes son las que instala el controlador ONOS por defecto, tal y como se ha observado en las pruebas del escenario anterior. A continuación, se analizan las 7 entradas:

- **Entradas 1 y 2**: Contienen las instrucciones de DROP para el tráfico, recibido en el puerto físico que comparte interfaz con el puerto local, cuya dirección MAC origen/destino es la MAC de la interfaz configurada como puerto local. Gracias a estas dos reglas, se evita que el `pipeline` del switch 1 procese su propio tráfico OpenFlow.
- **Entrada 3**: Flujo instalado para encaminar los mensajes ARP con destino el switch 2 cuando este hace un ARP REQUEST con la IP del controlador.
- **Entradas 4 y 5**: Contienen las instrucciones para encaminar el tráfico TCP de la conexión OpenFlow del switch 2 con el controlador.
- **Entradas 6 y 7**: Contienen las instrucciones para enviar al controlador el tráfico ARP y TCP de Openflow, y de este modo poder crear las reglas 3,4 y 5 que permiten que el switch 2 establezca la conexión OpenFlow con el controlador.

Gracias a estas entradas, el switch 1 es capaz de establecer el modo de control *in-band* y encaminar correctamente el tráfico necesario para que el switch 2 establezca la conexión con el controlador. Las entradas de la tabla de flujos del switch 2 serían similares a la del switch 1, exceptuando las entradas 4 y 5, que el switch 2 no las tendría porque no recibe tráfico de ningún otro dispositivo OpenFlow que funcione en modo *in-band* y que intente conectarse con el controlador. Por otro lado, la tabla de flujos del switch 3, que utiliza el modo de control *out-of-band*, sería similar a la obtenida en el escenario anterior para el switch 2 (Figura 5.11).

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:01"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="41", byte_cnt="12556", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:01"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="46", byte_cnt="6412", **insts=[]**},

3. match="oxm{**eth_dst="00:00:00:00:00:02", eth_type="0x806"**}", prio="65521", idle_to="10", hard_to="0", pkt_cnt="1", byte_cnt="60", **insts=[apply{acts=[out{port="2"}]}]**},

4. match="oxm{**in_port="2", eth_type="0x800", ipv4_src="10.0.0.102", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="43", byte_cnt="12718", **insts=[apply{acts=[out{port="1"}]}]**},

5. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.102", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="46", byte_cnt="8208", **insts=[apply{acts=[out{port="2"}]}]**},

6. match="oxm{**eth_type="0x806"**}", prio="65520", idle_to="0", hard_to="0", pkt_cnt="1", byte_cnt="60", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]**},

7. match="oxm{**eth_type="0x800", ip_proto="6", tcp_dst="6653"**}", prio="65520", idle_to="0", hard_to="0", pkt_cnt="1", byte_cnt="74", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]**},

8. match="oxm{**eth_type="0x8942"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="4", byte_cnt="560", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}, clear]**},

9. match="oxm{**eth_type="0x806"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="0", byte_cnt="0", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}, clear]**},

10. match="oxm{**eth_type="0x88cc"**}", prio="40000", idle_to="0", hard_to="0", pkt_cnt="4", byte_cnt="560", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}, clear]**},

11. match="oxm{**eth_type="0x800"**}", prio="5", idle_to="0", hard_to="0", pkt_cnt="0", byte_cnt="0", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}, clear]**}]}

Figura 5.17: Escenario 2 (ONOS): Entradas de la tabla de flujos del switch 1 tras iniciar el escenario

Tras haber analizado el estado inicial de los switches, se han lanzado 5 `pings` en cada sentido, entre el *host* 1 (IP `10.0.0.11`) y el *host* 2 (IP `10.0.0.12`). El resultado obtenido por línea de comandos se muestra en la Figura 5.18. Los dos *hosts* son capaces de comunicarse entre sí, por lo que se verifica que la red sigue funcionando correctamente aunque los switches operen en distintos modos de control, puesto que el switch 2, al que está conectado el *host* 1, utiliza el modo de control *in-band*, y el switch 3, al que está conectado el *host* 2, utiliza el modo de control *out-of-band*. Durante el intercambio de los `pings`

Se realizan los pings desde el **host 1**:
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
64 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=38.3 ms
64 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=0.439 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=0.756 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=0.470 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=0.428 ms

--- 10.0.0.12 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4059ms
rtt min/avg/max/mdev = 0.428/8.097/38.393/15.148 ms

Se realizan los pings desde el **host 2**:
PING 10.0.0.11 (10.0.0.11) 56(84) bytes of data.
64 bytes from 10.0.0.11: icmp_seq=1 ttl=64 time=0.443 ms
64 bytes from 10.0.0.11: icmp_seq=2 ttl=64 time=0.373 ms
64 bytes from 10.0.0.11: icmp_seq=3 ttl=64 time=0.522 ms
64 bytes from 10.0.0.11: icmp_seq=4 ttl=64 time=0.493 ms
64 bytes from 10.0.0.11: icmp_seq=5 ttl=64 time=0.480 ms

--- 10.0.0.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4027ms
rtt min/avg/max/mdev = 0.373/0.462/0.522/0.053 ms

Figura 5.18: Escenario 2 (ONOS): Resultados obtenidos mediante la herramienta ping

se ha capturado el tráfico en la interfaz `veth-ctrl`, que forma parte del enlace que comunica el switch 1 con el controlador, y en la interfaz de `loopback` para verificar si los mensajes PACKET_IN de cada switch se transmiten a través de la interfaz correcta. En la Figura 5.19 se puede verificar que los mensajes PACKET_IN originados en el switch 2, por el mensaje ARP REQUEST en el que el *host* 1 consulta la IP del *host* 2, y por los `pings` que envía el *host* 1 al *host* 2, llegan al controlador a través del enlace que comunica el controlador con el switch 1, que es el camino a seguir si se utiliza el modo de control *in-band*. Por otro lado, en la Figura 5.20 es posible comprobar que los mensajes PACKET_IN, generados por el mensaje ARP REQUEST en el que el *host* 2 consulta la IP del *host* 1 y por los `pings` que envía el *host* 2 al *host* 1, se transmiten al controlador a través de la interfaz de `loopback`, ya que el switch 3 utiliza el modo de control *out-of-band*.

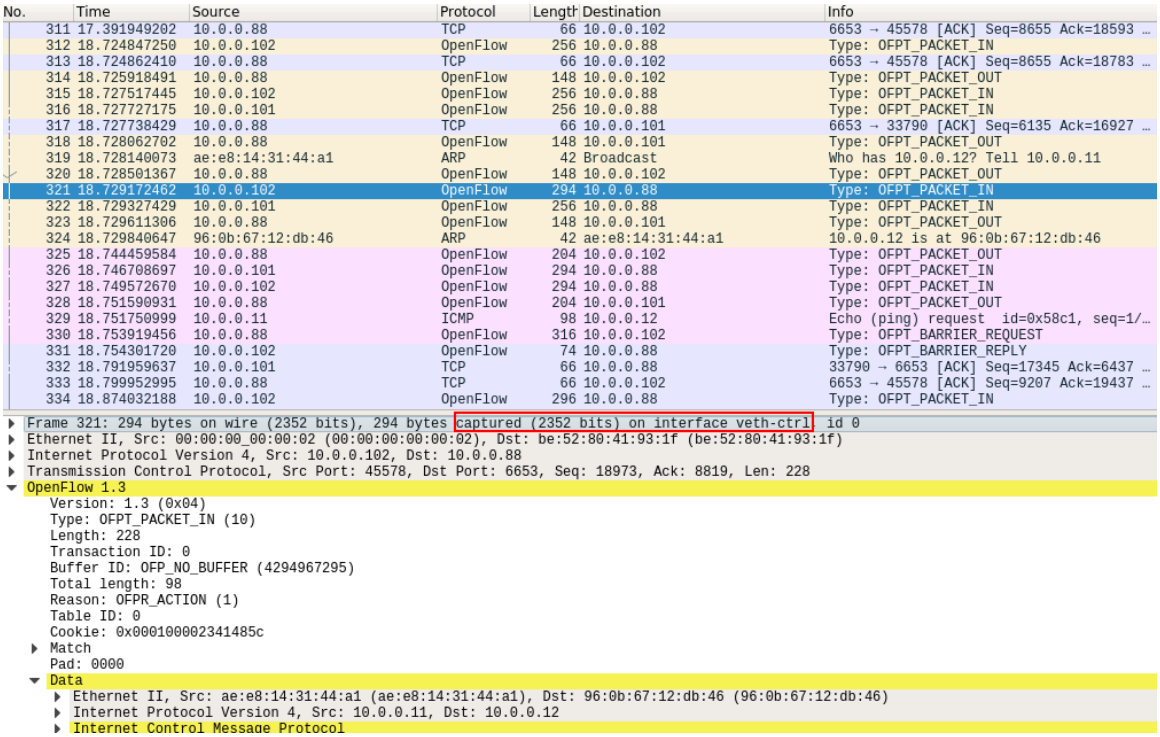


Figura 5.19: Escenario 2 (ONOS): Mensajes PACKET_IN generados por el switch 2 al lanzar los PINGS

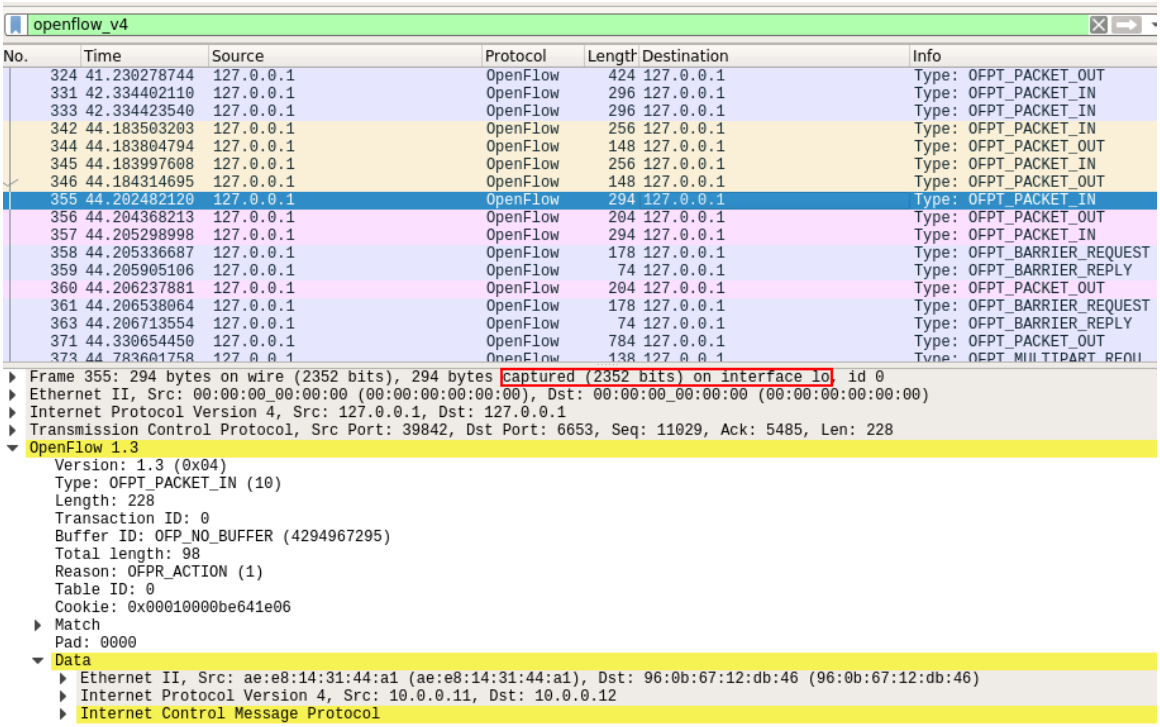


Figura 5.20: Escenario 2 (ONOS): Mensajes PACKET_IN generados por el switch 3 al lanzar los PINGS

Por último, en la Figura 5.21 se muestran las dos entradas que el controlador instala en la tabla de flujos del switch 2 con motivo de los `pings`, del mismo modo que ocurrió en el escenario anterior. Una vez se han realizado los `pings` y el controlador ha descubierto los dos *hosts*, la topología generada por el controlador se corresponde con la Figura 5.10, mostrada en el escenario anterior.

1. match="oxm{**in_port="3", eth_dst="ae:e8:14:31:44:a1", eth_src="96:0b:67:12:db:46"**}", prio="10", idle_to="0", hard_to="0", pkt_cnt="9", byte_cnt="882", **insts=[apply{acts=[out{port="2"}]}]**},

2. match="oxm{**in_port="2", eth_dst="96:0b:67:12:db:46", eth_src="ae:e8:14:31:44:a1"**}", prio="10", idle_to="0", hard_to="0", pkt_cnt="9", byte_cnt="882", **insts=[apply{acts=[out{port="3"}]}]**},

Figura 5.21: Escenario 2 (ONOS): Entradas de la tabla de flujos del switch 2 tras lanzar los pings entre los 2 hosts

5.3.2.2 Resultados utilizando RYU

Durante las pruebas realizadas en el escenario 2 utilizando el controlador RYU, se han obtenido varias capturas. En primer lugar, en la instantánea 5.22 de `Wireshark` se puede verificar que los mensajes de la conexión OpenFlow del switch 2 llegan al controlador a través de la interfaz `veth-ctrl`, que forma parte del enlace que comunica el switch 1 con el controlador. En ella se aprecian los mensajes ARP consultando la MAC del controlador y los mensajes TCP para establecer la conexión OpenFlow.

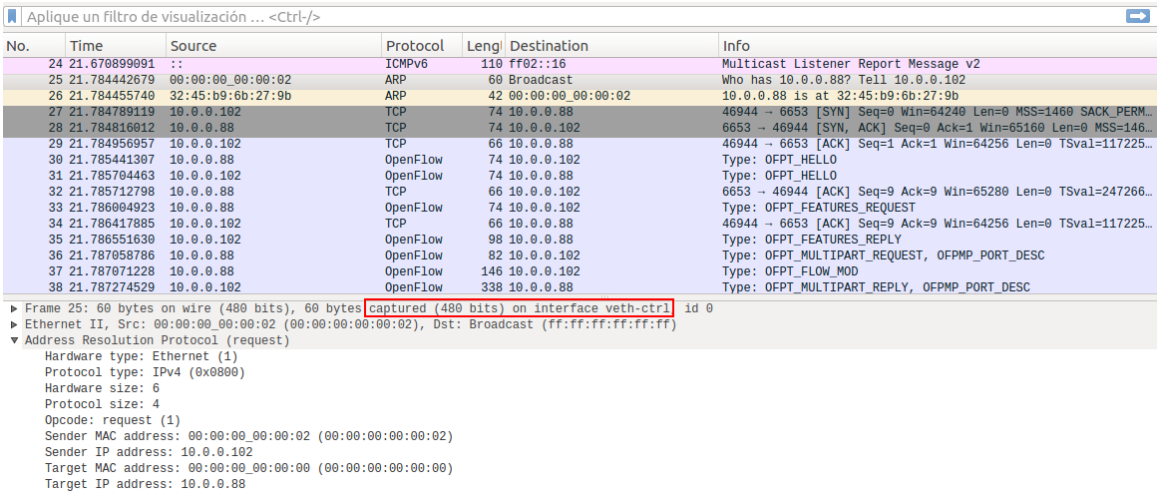


Figura 5.22: Escenario 2 (RYU): Mensajes de la conexión del switch 2 con el controlador en la interfaz veth-ctrl

Una vez se han iniciado todos los dispositivos del escenario 2, se han analizado los flujos que el switch 1 tiene instalados en su tabla de flujos. En la Figura 5.23 se muestran los flujos que se instalan debido al modo de control *in-band* del switch 1, y que ya se han explicado anteriormente en las pruebas realizadas con ONOS (Sección 5.3.2.1). En este caso, no se instalan los flujos para procesar el tráfico ARP y TCP como ocurre en las pruebas realizadas con ONOS (flujos 6 y 7 de la Figura 5.17). Se han omitido porque afectaban a los flujos que posteriormente RYU instalaba, provocando que la red no funcionara correctamente. No obstante, la lógica *in-band* no se ha visto afectada ya que la regla 6 de la Figura 5.23 engloba a los dos flujos anteriores.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:01"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="7", byte_cnt="710**", insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:01"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="7", byte_cnt="582", **insts=[]**},

3. match="oxm{**eth_dst="00:00:00:00:00:02", eth_type="0x806"**}", prio="65521", idle_to="10", hard_to="0", pkt_cnt="1", byte_cnt="60", **insts=[apply{acts=[out{port="2"}]}]**},

4. match="oxm{**in_port="2", eth_type="0x800", ipv4_src="10.0.0.102", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="7", byte_cnt="1078", **insts=[apply{acts=[out{port="1"}]}]**},

5. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.102", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="8", byte_cnt="648", **insts=[apply{acts=[out{port="2"}]}]**},

6. match="oxm{**all match**}", prio="0", idle_to="0", hard_to="0", pkt_cnt="2", byte_cnt="134", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]}]**}

Figura 5.23: Escenario 2 (RYU): Entradas de la tabla de flujos del switch 1 tras iniciar el escenario

A continuación, se ha verificado la comunicación entre los dos *hosts* desplegados en el escenario mediante la herramienta `ping`. Para ello, se han lanzado 5 `pings` en cada sentido y el resultado obtenido se muestra en la Figura 5.24.

Se realizan los pings desde el **host 1**:
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
64 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=19.3 ms
64 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=0.410 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=0.950 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=0.745 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=0.437 ms

--- 10.0.0.12 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4071ms
rtt min/avg/max/mdev = 0.410/4.386/19.389/7.504 ms

Se realizan los pings desde el **host 2**:
PING 10.0.0.11 (10.0.0.11) 56(84) bytes of data.
64 bytes from 10.0.0.11: icmp_seq=1 ttl=64 time=0.453 ms
64 bytes from 10.0.0.11: icmp_seq=2 ttl=64 time=0.660 ms
64 bytes from 10.0.0.11: icmp_seq=3 ttl=64 time=0.659 ms
64 bytes from 10.0.0.11: icmp_seq=4 ttl=64 time=0.578 ms
64 bytes from 10.0.0.11: icmp_seq=5 ttl=64 time=0.576 ms

--- 10.0.0.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 0.453/0.585/0.660/0.077 ms

Figura 5.24: Escenario 2 (RYU): Resultados obtenidos mediante la herramienta ping

Tras comprobar que ambos dispositivos son capaces de comunicarse entre sí, se han analizado las capturas del intercambio de paquetes producidos por estos `pings`, obtenidas mediante `Wireshark`. En la imagen 5.25 aparecen los mensajes PACKET_IN que el switch 2 genera por la consulta ARP, que el *host* 1 realiza acerca de la IP `10.0.0.12`, dirección IP del *host* 2, y por el mensajes ICMP generado por la herramienta `ping`. Como consecuencia de la difusión del ARP REQUEST el switch 1 también genera un mensaje PACKET_IN (mensaje 72) aunque este no es relevante. Del mismo modo, en la Figura 5.26 se muestran los mensajes PACKET_IN generados por el switch 3 y enviados al controlador a través de la interfaz de `loopback`, ya que este switch funciona con el modo de control *out-of-band*. Verificando de esta manera que los switches intercambian los mensajes con el controlador a través de las interfaces correctas, según el tipo de control que tenga configurado cada dispositivo.

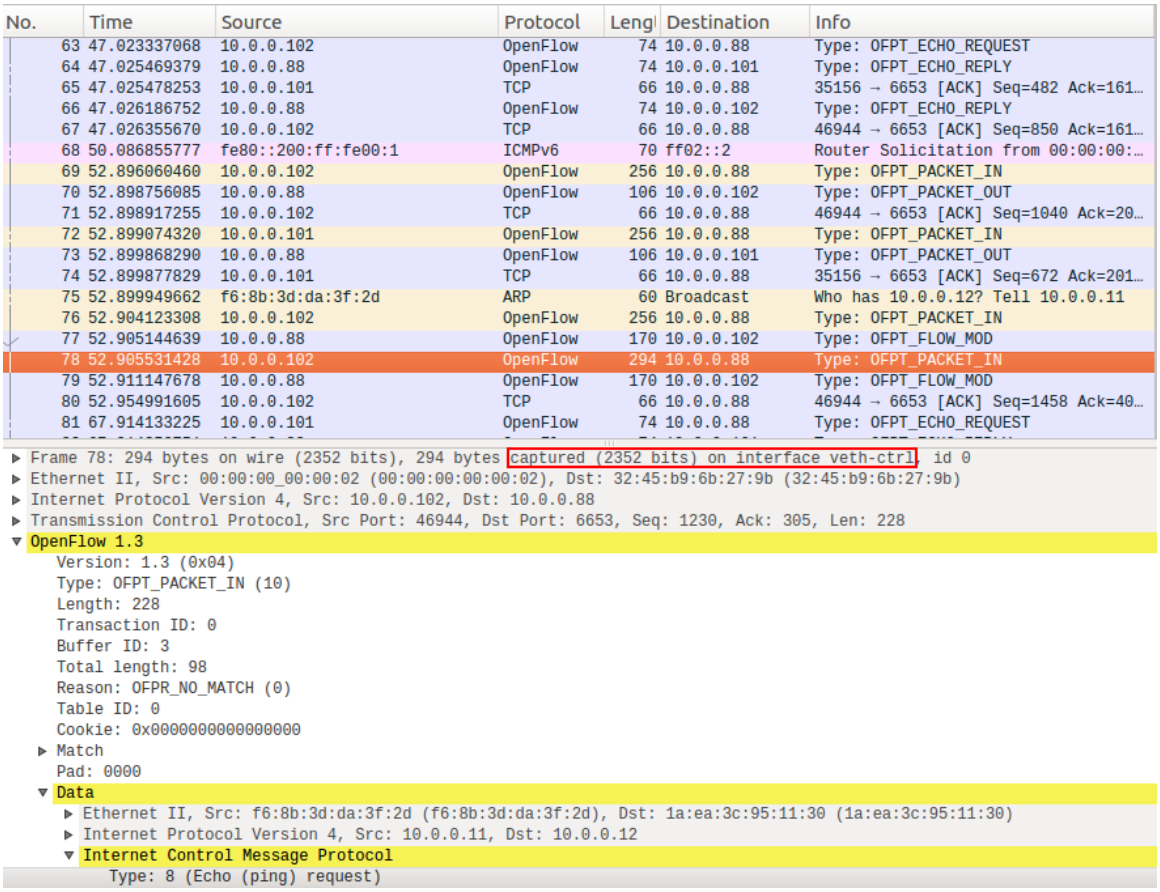


Figura 5.25: Escenario 2 (RYU): Mensajes PACKET_IN generados por el switch 2 al lanzar los pings

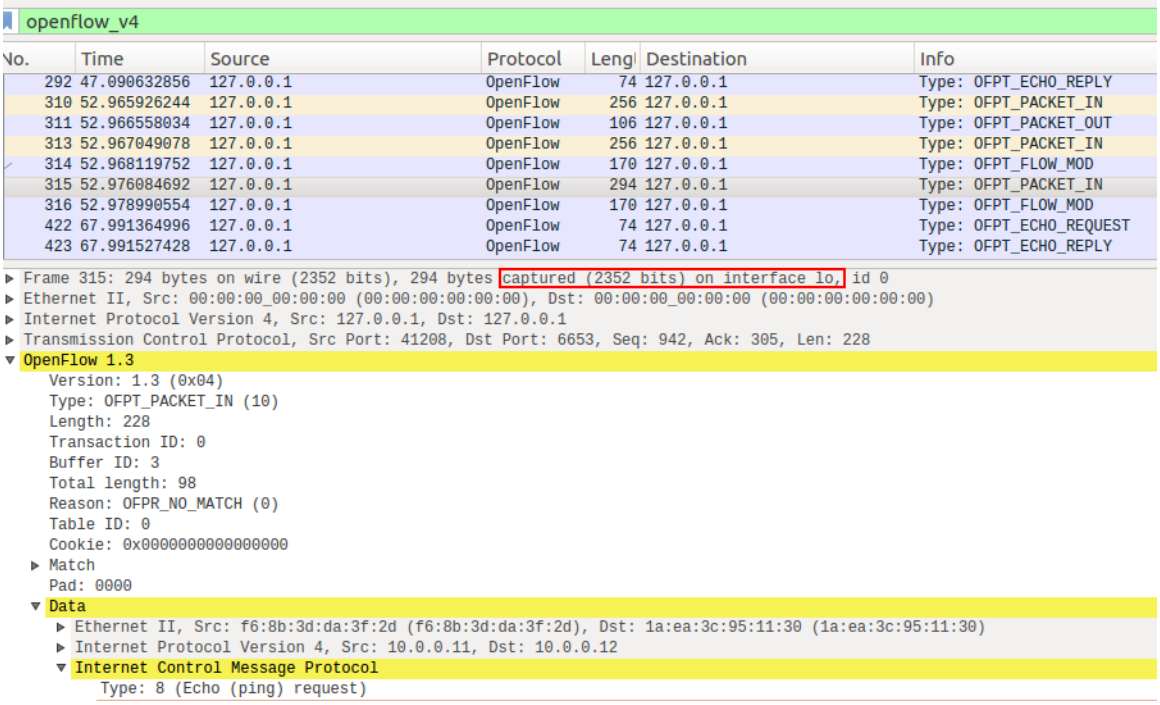


Figura 5.26: Escenario 2 (RYU): Mensajes PACKET_IN generados por el switch 3 al lanzar los PINGS

Por último, como consecuencia de este intercambio de mensajes, RYU instala dos reglas en los switches 2 y 3 para encaminar correctamente los paquetes entre los *hosts* 1 y 2. En la Figura 5.27 se muestran únicamente las reglas instaladas en el switch 2 ya que las del switch 3 tienen la misma estructura, variando los puertos de entrada y el de salida. El flujo 3 sirve para encaminar el tráfico en sentido *host* 2 a *host* 1 y el flujo 4 lo hace en sentido contrario.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:02"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="18", byte_cnt="2653", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:02"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="15", byte_cnt="1414", **insts=[]**},

3. match="oxm{**in_port="2", eth_dst="f6:8b:3d:da:3f:2d", eth_src="1a:ea:3c:95:11:30"**}", prio="1", idle_to="0", hard_to="0", pkt_cnt="12", byte_cnt="1040", **insts=[apply{acts=[out{port="3"}]}]**},

4. match="oxm{**in_port="3", eth_dst="1a:ea:3c:95:11:30", eth_src="f6:8b:3d:da:3f:2d"**}", prio="1", idle_to="0", hard_to="0", pkt_cnt="11", byte_cnt="980", **insts=[apply{acts=[out{port="2"}]}]**},

5. match="oxm{**all match**}", prio="0", idle_to="0", hard_to="0", pkt_cnt="4", byte_cnt="305", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]}]**}

Figura 5.27: Escenario 2 (RYU): Entradas de la tabla de flujos del switch 2 tras lanzar los pings entre los 2 hosts

5.3.3 Pruebas realizadas en el escenario 3: Topología formada por 4 switches en modo de control in-band

Las pruebas que se han realizado en las dos topologías que conforman el tercer escenario tienen como objetivo comprobar si funcionan correctamente las implementaciones del modo de control *in-band* y la reconfiguración de un nuevo puerto de control haciendo uso de los caminos generados por el protocolo Amaru. Para realizar las pruebas en este entorno se ha utilizado únicamente el controlador ONOS.

En la topología con nivel de conectividad 2 (Figura 5.4), en primer lugar, se ha comprobado si los paquetes OpenFlow de cada switch se transmiten a través de la interfaz que tienen configurada como puerto de control y si los *hosts* pueden comunicarse entre sí. A continuación, se ha deshabilitado el enlace que comunica el switch 1 con el switch 3, como aparece reflejado en la Figura 5.28, se ha examinado si el switch 3 reconfigura su puerto de control y se ha verificado si los paquetes OpenFlow se transmiten al controlador a través del switch 4 y del switch 2, dado que no existiría otro camino posible hasta el controlador. Además, una vez hecha la reconfiguración, se ha comprobado si existe comunicación entre los *hosts*.

En la topología con nivel de conectividad 3 (Figura 5.5) se ha procedido del mismo modo que con la topología con conectividad 2. Tras comprobar que los paquetes OpenFlow se transmiten a través de las interfaces inicialmente configuradas como puertos de control y que los *hosts* pueden comunicarse entre ellos, se ha deshabilitado la interfaz que une el switch 3 con el switch 1, como aparece reflejado en la Figura 5.29, se ha verificado si el switch 3 ha reconfigurado un nuevo puerto y, a continuación, se ha comprobado si los paquetes OpenFlow se transmiten a través de los switches que forman parte del nuevo camino hasta el controlador y si los *hosts* siguen manteniendo la comunicación. Seguidamente, se ha comprobado cuál es el nuevo puerto local del switch 3 y, en función de la interfaz configurada, se ha deshabilitado el enlace del nuevo puerto local para observar el comportamiento del switch. Por último, se ha comprobado si los *hosts* mantienen la capacidad de comunicarse unos con otros mediante la herramienta `ping`.

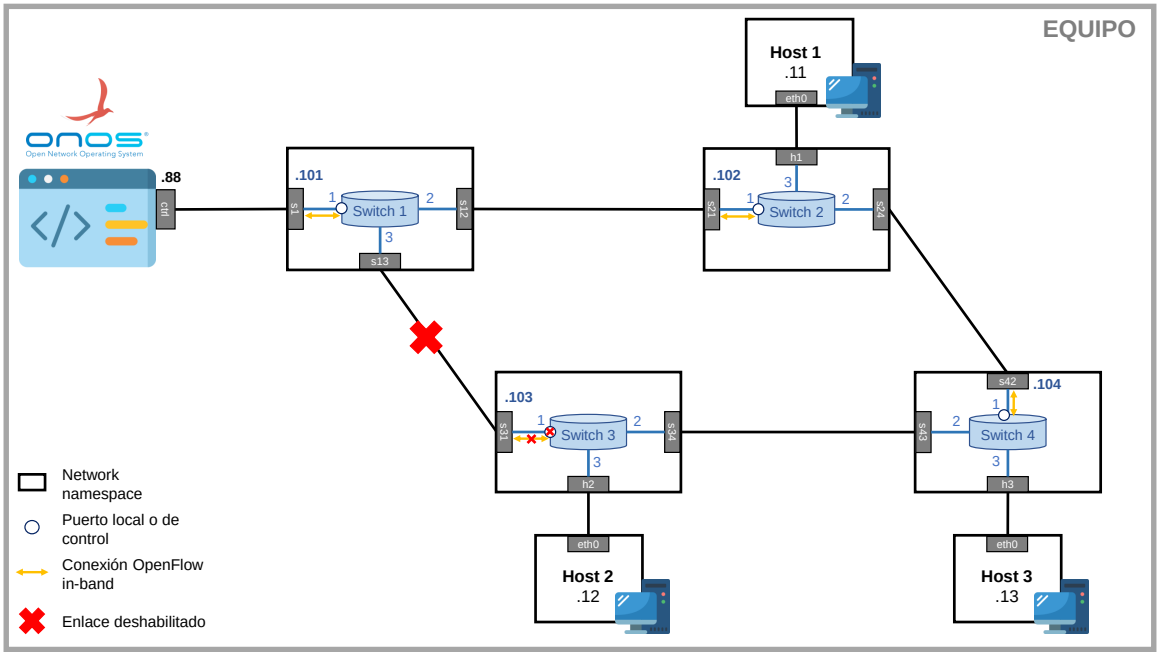


Figura 5.28: Escenario 3 con nivel de conectividad 2 al deshabilitar un enlace

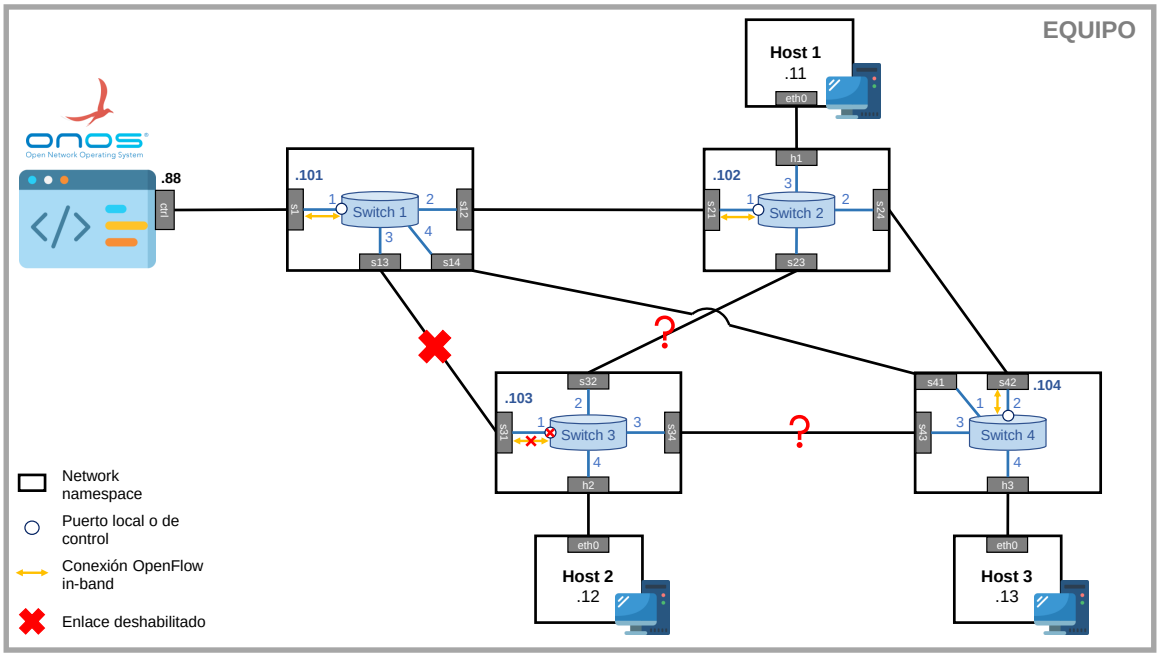


Figura 5.29: Escenario 3 con nivel de conectividad 3 al deshabilitar un enlace

Para deshabilitar un enlace creado mediante un dispositivo `veth` es necesario desactivar uno de los dos extremos:

Código 5.4: Comando deshabilitar una interfaz

```
#Deshabilitar la interfaz de un network namespace desde el network namespace
ip link set [nombre_interfaz] down

#Deshabilitar la interfaz de un network namespace desde el root
sudo ip netns exec [nombre_network_ns] ip link set [nombre_interfaz] down
```

5.3.3.1 Resultados utilizando la topología con nivel de conectividad 2

En este escenario se han realizado las primeras pruebas para verificar la reconfiguración del puerto local del switch BOFUSS mediante el protocolo Amaru. Antes de llevar a cabo ninguna prueba, se han inspeccionado los flujos que el switch 1 tiene instalados inicialmente. En la Figura 5.30 se puede comprobar que el switch 1 ha instalado las reglas necesarias para encaminar el tráfico OpenFlow del resto de switches del entorno. Además, se ha verificado, mediante la herramienta `ping`, que los *hosts* desplegados en el escenario pueden comunicarse entre sí.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:01"**}", idle_to="0", hard_to="0", pkt_cnt="78", byte_cnt="27632", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:01"**}", idle_to="0", hard_to="0", pkt_cnt="78", byte_cnt="16296", **insts=[]**},

3. match="oxm{**in_port="2", eth_type="0x800", ipv4_src="10.0.0.102", ipv4_dst="10.0.0.88", ip_proto="6"**}", idle_to="5", hard_to="0", pkt_cnt="72", byte_cnt="19932", **insts=[apply{acts=[out{port="1"}]}]**},

4. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.102", ip_proto="6"**}", idle_to="5", hard_to="0", pkt_cnt="69", byte_cnt="12192", **insts=[apply{acts=[out{port="2"}]}]**},

5. match="oxm{**in_port="3", eth_type="0x800", ipv4_src="10.0.0.103", ipv4_dst="10.0.0.88", ip_proto="6"**}", idle_to="5", hard_to="0", pkt_cnt="77", byte_cnt="18852", **insts=[apply{acts=[out{port="1"}]}]**},

6. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.103", ip_proto="6"**}", idle_to="5", hard_to="0", pkt_cnt="72", byte_cnt="10690", **insts=[apply{acts=[out{port="3"}]}]**},

7. match="oxm{**in_port="2", eth_type="0x800", ipv4_src="10.0.0.104", ipv4_dst="10.0.0.88", ip_proto="6"**}", idle_to="5", hard_to="0", pkt_cnt="76", byte_cnt="18694", **insts=[apply{acts=[out{port="1"}]}]**},

8. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.104", ip_proto="6"**}", idle_to="5", hard_to="0", pkt_cnt="79", byte_cnt="11818", **insts=[apply{acts=[out{port="2"}]}]**},

Figura 5.30: Escenario 3 (N=2): Contenido de la tabla de flujos del switch 1

Por último, se ha comprobado que se ha completado la exploración de Amaru y que los switches han rellenado sus tablas con los caminos generados mediante la exploración. Para ello, se han consultado los *logs* que genera cada switch del contenido de su tabla de Amaru. A modo de ejemplo, en la Tabla 5.1 se muestra el contenido de la tabla Amaru del switch 4 una vez ha finalizado la exploración. Teniendo presente la topología utilizada (Figura 5.4), en la tabla aparecen los dos caminos posibles de los que dispone el switch 4 para alcanzar el controlador. El camino 1 refleja el camino superior, que se inicia a través de su puerto 1 y que permite alcanzar el controlador en dos saltos, es decir, el tráfico de control atravesaría el switch 2 y, por último, el switch 1, que es el nodo raíz. Del mismo modo, el camino 2 representa el camino inferior, que se inicia a través de su puerto 2 y atraviesa el switch 3 y el switch 1.

Posición	AMAC	Nivel	Puerto	Activa
1	1:2:2	3	1	1
2	1:3:2	3	2	1

Tabla 5.1: Escenario 3 (N=2): Contenido inicial de la tabla Amaru del switch 4

Tal y como se ha visto en el análisis del switch BOFUSS, cada puerto está asociado a una interfaz del `network namespace` en el que se inicia. Por lo tanto, al escoger un camino de la tabla de Amaru, el puerto asociado se utiliza para identificar la interfaz que se configurará como puerto de control.

Una vez se ha comprobado que la configuración inicial del escenario es correcta, se ha consultado la interfaz gráfica del controlador ONOS para verificar que la topología es la correcta (Figura 5.31).

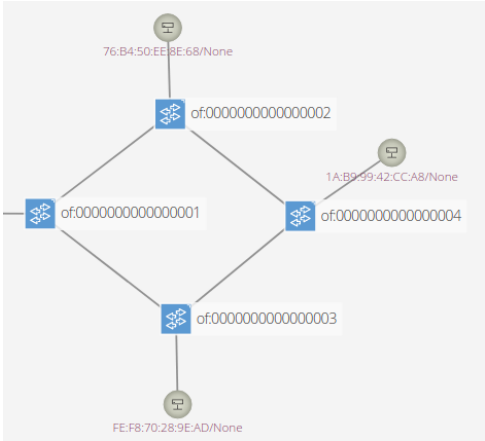


Figura 5.31: Escenario 3 (N=2): Topología inicial en la interfaz gráfica de ONOS

Tras inicializar el escenario, se ha analizado el contenido de la tabla de Amaru del switch 3 (tabla 5.2) con la intención de comprobar qué caminos están disponibles y tomar constancia de las interfaces que pueden configurarse como puerto local. En ella se puede observar que existen dos caminos disponibles. El primer camino está asociado al puerto 1. Este camino permite alcanzar el controlador a través del switch 1. El segundo camino está asociado al puerto número 2 y permite alcanzar el controlador a través de los switches 4, 2 y 1 de la topología.

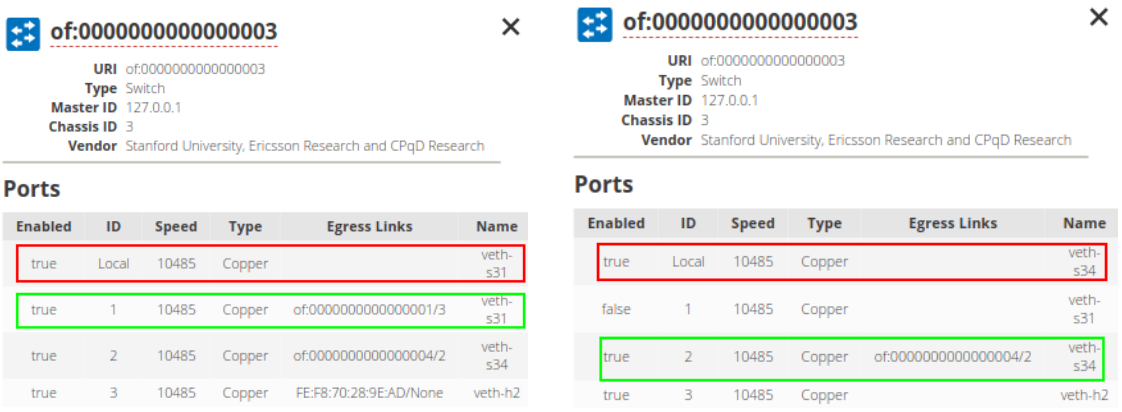
Posición	AMAC	Nivel	Puerto	Activa
1	1:3	2	1	1
2	1:2:2:2	4	2	1

Tabla 5.2: Escenario 3 (N=2): Caminos generados en la tabla de Amaru del switch 3

Posición	AMAC	Nivel	Puerto	Activa
1	1:3	2	1	0
2	1:2:2:2	4	2	1

Tabla 5.3: Escenario 3 (N=2): Caminos en la tabla de Amaru del switch 3 tras deshabilitar el enlace con el switch 1

Si se analiza la Figura 5.32a es posible comprobar que la interfaz del puerto número 1 coincide con la interfaz que tiene configurada el puerto local. Una vez se ha analizado el estado inicial del switch 3, se ha utilizado el comando mencionado en el bloque de Código 5.4 para deshabilitar el enlace que comunica el switch 3 con el switch 1, y por lo tanto, las interfaces `veth-s31` y `veth-s13`, tal y como se muestra en la Figura 5.28. Una vez ejecutado el comando, los caminos de Amaru disponibles se muestran en la Tabla 5.3, donde se puede comprobar que el primer camino, asociado al puerto 1, está deshabilitado, puesto que su interfaz se ha desactivado, y que el único camino restante es el asociado al puerto 2 del switch. Si se analiza la nueva lista de puertos del switch 3 que muestra ONOS (Figura 5.32b), se puede verificar que el nuevo puerto local especifica la interfaz del puerto con identificador 2. Por lo tanto, los caminos disponibles de los generados por Amaru y el nuevo puerto local configurado coinciden.



(a) Puerto local inicial del switch 3 (b) Nuevo puerto local configurado del switch 3

Figura 5.32: Escenario 3 (N=2): Puerto local del switch 3 antes y después de reconfigurar uno nuevo

De la misma manera, en la Figura 5.33 se muestran las entradas de DROP que el switch 3 tiene instaladas para no procesar su propio tráfico OpenFlow. En la parte superior aparecen las entradas DROP para el puerto de control configurado inicialmente, y en la parte inferior aparecen las entradas DROP para el nuevo puerto de control configurado. De este modo se verifica nuevamente que la interfaz del puerto con identificador 2 se ha configurado como puerto de control, según estaba previsto teniendo en cuenta el contenido de la tabla generada por el protocolo Amaru.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:03"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="170", byte_cnt="48554", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:03"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="182", byte_cnt="27546", **insts=[]**},

---

1. match="oxm{**in_port="2", eth_src="00:00:00:00:00:03"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="33", byte_cnt="7987", **insts=[]**},

2. match="oxm{**in_port="2", eth_dst="00:00:00:00:00:03"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="29", byte_cnt="4826", **insts=[]**},

Figura 5.33: Escenario 3 (N=2): Contenido de la tabla de flujos del switch 3 antes y después de reconfigurar el puerto local

Después de comprobar que el switch 3 ha reconfigurado su puerto local, se ha verificado que el resto de switches de la topología que se han visto afectados por el tráfico del switch 3, se han reconfigurado para encaminar correctamente el nuevo tráfico. Como muestra de ello, se proporciona la Figura 5.34, que refleja contenido de la tabla del switch 4, donde puede comprobarse que el switch ha instalado las reglas necesarias para encaminar correctamente los paquetes de la conexión OpenFlow del switch 3. No obstante, aunque no se hayan incluido pruebas gráficas, se ha hecho esta comprobación con las tablas de flujos de todos los switches y se ha verificado que todos se han reconfigurado correctamente.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="258", byte_cnt="72258", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="274", byte_cnt="43490", **insts=[]**},

3. match="oxm{**eth_dst="00:00:00:00:00:03", eth_type="0x806"**}", prio="65521", idle_to="10", hard_to="0", pkt_cnt="1", byte_cnt="60", **insts=[apply{acts=[out{port="2"}]}]**},

4. match="oxm{**in_port="2", eth_type="0x800", ipv4_src="10.0.0.103", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="31", byte_cnt="7358", **insts=[apply{acts=[out{port="1"}]}]**},

5. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.103", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="28", byte_cnt="4766", **insts=[apply{acts=[out{port="2"}]}]**},

Figura 5.34: Escenario 3 (N=2): Contenido de la tabla de flujos del switch 4 tras la reconfiguración del puerto local del switch 3

Antes de finalizar las pruebas, se ha comprobado mediante la herramienta `ping` que los *hosts* siguen manteniendo la comunicación tras la reconfiguración del puerto local del switch 3. Debido a lo extenso de la salida por línea de comandos generada por la herramienta `ping` se ha decidido no incluir la captura en el documento. A continuación, se ha consultado la topología final representada en la interfaz gráfica de ONOS para verificar si ha percibido los cambios realizados en los enlaces (Figura 5.35). En ella puede comprobarse que el switch 3 y el switch 1 ya no están directamente comunicados. Asimismo, se verifica que el camino que deben seguir los paquetes OpenFlow entre el switch 3 y el controlador es a través del switch 2 y del switch 4.

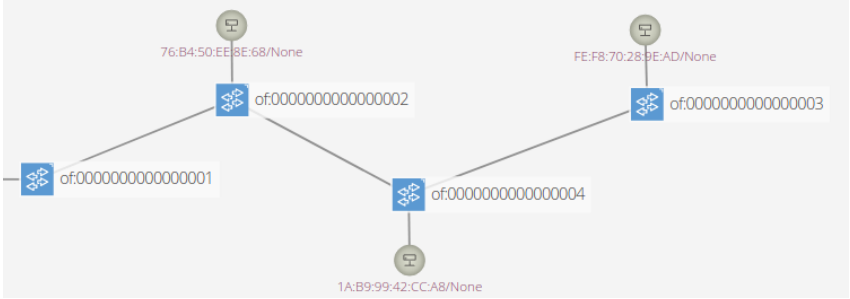


Figura 5.35: Escenario 3 (N=2): Topología final obtenida en la interfaz gráfica de ONOS

5.3.3.2 Resultados utilizando la topología con nivel de conectividad 3

A lo largo de esta sección se describen los resultados obtenidos mediante las pruebas realizadas en la topología con un nivel de conectividad entre switches igual a 3. Antes de poner a prueba el funcionamiento de Amaru y la reconfiguración del puerto local de los switches, se ha verificado mediante la herramienta `ping` que los *hosts* son capaces de comunicarse entre sí y se ha comprobado que la topología que genera ONOS (Figura 5.36) se corresponde con la del escenario.

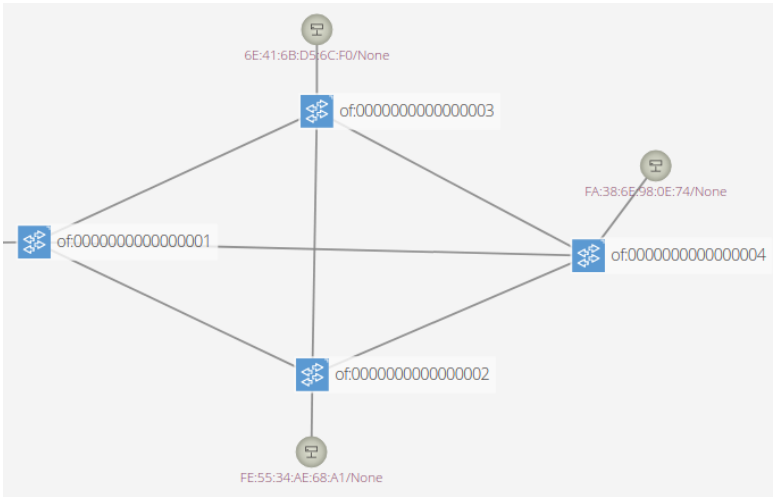


Figura 5.36: Escenario 3 (N=3): Topología inicial en la interfaz gráfica de ONOS

Además, se ha analizado el contenido de la tabla de Amaru del switch 3 (Tabla 5.4). Esta contiene un total de 5 caminos disponibles, asociados a 3 puertos del switch. Asímismo, se ha observado la lista de puertos que el switch le ha comunicado a ONOS (Figura 5.37a) y se ha comprobado que la interfaz del puerto local coincide con la interfaz del puerto número 1.

Posición	AMAC	Nivel	Puerto	Activa
1	1:3	2	1	1
2	1:2:2	3	2	1
3	1:4:3	3	3	1
4	1:4:2:2	4	2	1
5	1:2:3:3	4	3	1

Tabla 5.4: Escenario 3 (N=3): Estado inicial de la tabla de Amaru del switch 3

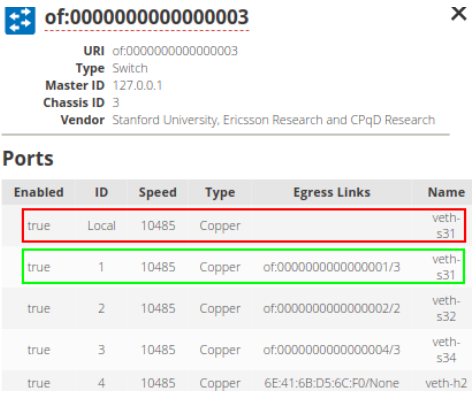
Posición	AMAC	Nivel	Puerto	Activa
1	1:3	2	1	0
2	1:2:2	3	2	1
3	1:4:3	3	3	1
4	1:4:2:2	4	2	1
5	1:2:3:3	4	3	1

Tabla 5.5: Escenario 3 (N=3): Contenido de la tabla de Amaru del switch 3 tras reconfigurar una vez su puerto local

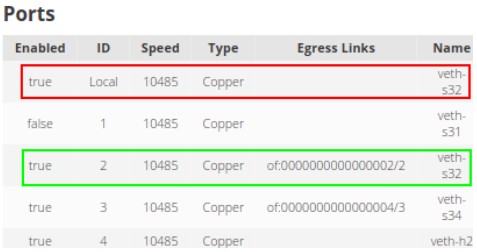
Posición	AMAC	Nivel	Puerto	Activa
1	1:3	2	1	0
2	1:2:2	3	2	0
3	1:4:3	3	3	1
4	1:4:2:2	4	2	0
5	1:2:3:3	4	3	1

Tabla 5.6: Escenario 3 (N=3): Contenido de la tabla de Amaru del switch 3 tras reconfigurar dos veces su puerto local

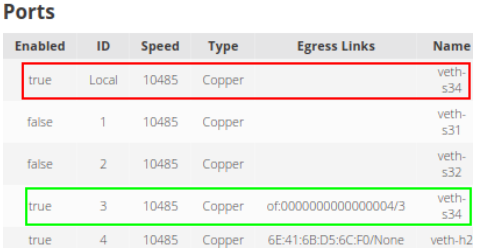
Una vez analizado el estado inicial del entorno, se han llevado a cabo las pruebas previstas. En primer lugar, se ha deshabilitado el enlace que une el switch 1 y el switch 3. Como consecuencia de ello, se han deshabilitado las interfaces `veth-s13` y `veth-s31`. Para verificar que el switch 3 ha reconfigurado su puerto local, se ha consultado la lista de puertos que el switch 3 le ha comunicado al controlador ONOS (Figura 5.37b). En esta lista aparece una nueva interfaz configurada como puerto local que se corresponde con la interfaz del puerto número 2 del switch. Consultando el nuevo estado de la tabla de Amaru (Tabla 5.5) se puede verificar que el puerto número 2 está asociado al segundo camino disponible ya que el primero ya no está activo al haberse deshabilitado la interfaz de este puerto. Antes de continuar con la siguiente prueba, se ha revisado el contenido de la tabla de flujos del switch 2 para comprobar que ha instalado los flujos necesarios para encaminar correctamente el tráfico OpenFlow del switch 3.



(a) Estado inicial de los puertos del switch 3



(b) Estado de los puertos del switch 3 tras la primera reconfiguración



(c) Estado de los puertos del switch 3 tras la segunda reconfiguración

Figura 5.37: Escenario 3 (N=3): Evolución de los puertos del switch 3

En la Figura 5.38 se muestra contenido de la tabla de flujos del switch 2 después de que el switch 3 reconfigure su puerto de control, en ella se han remarcado las 3 reglas que el switch 2 ha instalado para poder encaminar el flujo OpenFlow del switch 3. El flujo número 5 permite encaminar el ARP REPLY que el controlador le envía al switch 3, proporcionándole la MAC de la IP que el switch ha consultado y los flujos 6 y 7 sirven para encaminar el tráfico TCP propio de la conexión OpenFlow entre el controlador y el switch 3.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:02"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="262", byte_cnt="84016", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:02"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="284", byte_cnt="49095", **insts=[]**},

3. match="oxm{**in_port="3", eth_type="0x800", ipv4_src="10.0.0.104", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="249", byte_cnt="76256", insts=[apply{acts=[out{port="1"}]}]},

4. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.104", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="252", byte_cnt="45337", **insts=[apply{acts=[out{port="3"}]}]**},

5. match="oxm{**eth_dst="00:00:00:00:00:03", eth_type="0x806"**}", prio="65521", idle_to="10", hard_to="0", pkt_cnt="1", byte_cnt="60", **insts=[apply{acts=[out{port="2"}]}]**},

6. match="oxm{**in_port="2", eth_type="0x800", ipv4_src="10.0.0.103", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="36", byte_cnt="8833", **insts=[apply{acts=[out{port="1"}]}]**},

7. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.103", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="35", byte_cnt="6680", **insts=[apply{acts=[out{port="2"}]}]**},

Figura 5.38: Escenario 3 (N=3): Conetenido de la tabla de flujos del switch 2 tras la primera reconfiguración del puerto local del switch 3

En segundo lugar, se ha deshabilitado el enlace que comunica el switch 2 con el switch 3 para obligar a este último a que vuelva a reconfigurar su puerto local porque, tal y como se ha visto anteriormente, el nuevo puerto local se ha configurado en la interfaz asociada al puerto 2 del switch 3, que pertenece al enlace que lo comunica con el switch 2. Para comprobar si el switch 3 ha configurado un nuevo puerto local, se ha examinado nuevamente la lista de puertos del switch 3 disponible en el controlador ONOS (Figura 5.37c). En esta lista se puede apreciar que el nuevo puerto local se ha configurado en la interfaz del puerto número 3, puerto a través del cual se comunica con el switch 4. Además, las interfaces que formaban parte de los enlaces desactivados se muestran como deshabilitadas.

Seguidamente, se ha inspeccionado la tabla de flujos del switch 4 y se ha verificado que ha instalado los flujos necesarios para encaminar el tráfico OpenFlow del switch 3 (Figura 5.39). Por último, se debe mencionar que el estado final de la tabla de Amaru del switch 3 (Tabla 5.6) también refleja que se han deshabilitado las interfaces de los puertos 1 y 2 del switch ya que los caminos asociados a estos aparecen desactivados.

1. match="oxm{**in_port="2", eth_src="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="404", byte_cnt="119612", **insts=[]**},

2. match="oxm{**in_port="2", eth_dst="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="402", byte_cnt="71244", **insts=[]**},

3. match="oxm{**eth_dst="00:00:00:00:00:03", eth_type="0x806"**}", prio="65521", idle_to="10", hard_to="0", pkt_cnt="1", byte_cnt="60", **insts=[apply{acts=[out{port="3"}]}]**},

4. match="oxm{**in_port="3", eth_type="0x800", ipv4_src="10.0.0.103", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="41", byte_cnt="9100", **insts=[apply{acts=[out{port="2"}]}]**},

5. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.103", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="40", byte_cnt="5930", **insts=[apply{acts=[out{port="3"}]}]**},

6. match="oxm{**eth_type="0x806"**}", prio="65520", idle_to="0", hard_to="0", pkt_cnt="9", byte_cnt="540", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]**},

7. match="oxm{**eth_type="0x800", ip_proto="6", tcp_dst="6653"**}", prio="65520", idle_to="0", hard_to="0", pkt_cnt="1", byte_cnt="569", **insts=[apply{acts=[out{port="ctrl", mlen="65535"}]}]**},

Figura 5.39: Escenario 3 (N=3): Conetenido de la tabla de flujos del switch 4 tras la segunda reconfiguración del puerto local del switch 3

Antes de dar por concluidas las pruebas, se ha confirmado mediante la herramienta `ping` que los *hosts* mantienen la capacidad de comunicarse. La topología final que recoge la interfaz gráfica de ONOS aparece en la Figura 5.40, donde puede comprobarse que el switch 3 ha dejado de estar directamente comunicado con los switchs 1 y 2, hecho totalmente esperado después de haber deshabilitado esos dos enlaces.

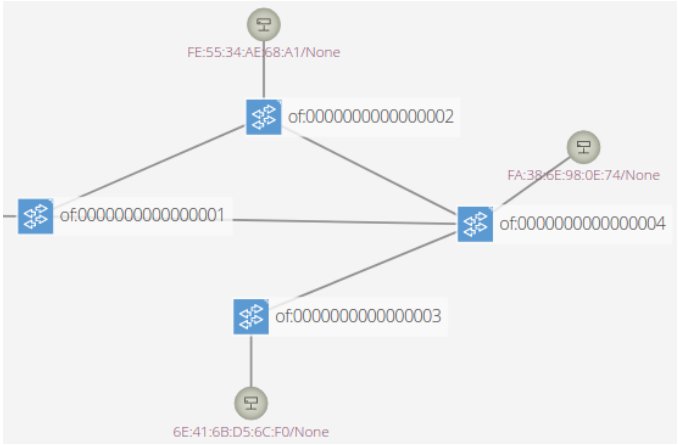


Figura 5.40: Escenario 3 (N=3): Topología final en la interfaz gráfica de ONOS

5.3.4 Pruebas realizadas en el escenario 4: Topología en forma de triple rombo funcionando en modo de control in-band

Las pruebas realizadas en el escenario 4 tienen el objetivo de verificar si la red formada por múltiples switches operando con el modo de control *in-band* se mantiene estable cuando varios de estos switches son forzados a reconfigurar su puerto local. Para realizar las pruebas se ha utilizado el controlador ONOS.

En primer lugar, se ha comprobado que cada switch se comunica con el controlador a través de la interfaz configurada inicialmente como puerto de control. A continuación, se ha deshabilitado el enlace que comunica el switch 5 con el switch 2 y se ha comprobado si el switch 5 ha reconfigurado su puerto local y si el camino que sigue su tráfico OpenFlow se ha redirigido a través del switch 4. Tras realizar estas comprobaciones, se ha deshabilitado el enlace que une el switch 6 con el switch 3 y se ha comprobado si su tráfico de control se ha redirigido a través del switch 4. Por último, se ha deshabilitado el enlace que comunica el switch 4 con el switch 2 y se ha verificado si se ha reconfigurado un nuevo puerto de control y si su tráfico OpenFlow se ha reconducido a través del mismo. Todas las modificaciones que han sufrido los enlaces quedan reflejadas en la Figura 5.41.

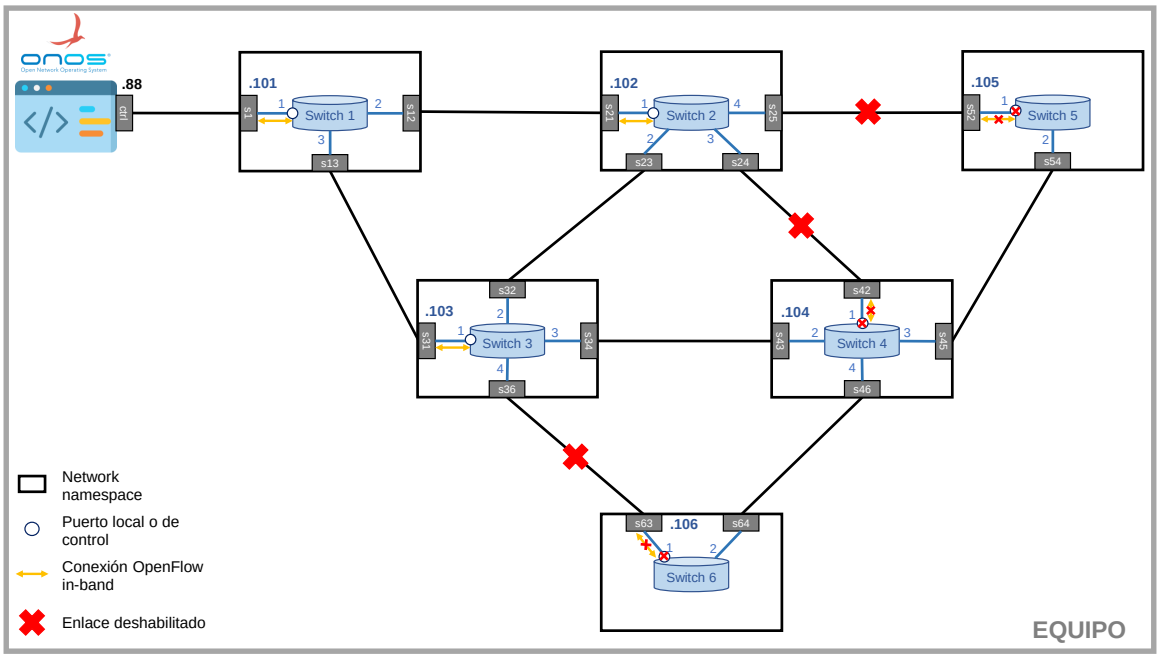


Figura 5.41: Representación de los enlaces que se deshabilitan en el escenario 4

5.3.4.1 Resultados

En este aparatado se describen los resultados obtenidos al realizar las pruebas acordadas para el último de los escenarios. Del mismo modo que se ha hecho en anteriores escenarios, antes de mostrar los resultados, se ha comprobado que los switches han establecido la conexión OpenFlow con el controlador a través de la interfaz que se ha configurado inicialmente como puerto de control. Una vez hecha esta verificación, se ha examinado la topología que muestra ONOS en su interfaz gráfica (Figura 5.42) y se ha confirmado que se corresponde con la diseñada en el escenario. Tras verificar que el escenario se ha iniciado correctamente, se han llevado a cabo las pruebas previstas.

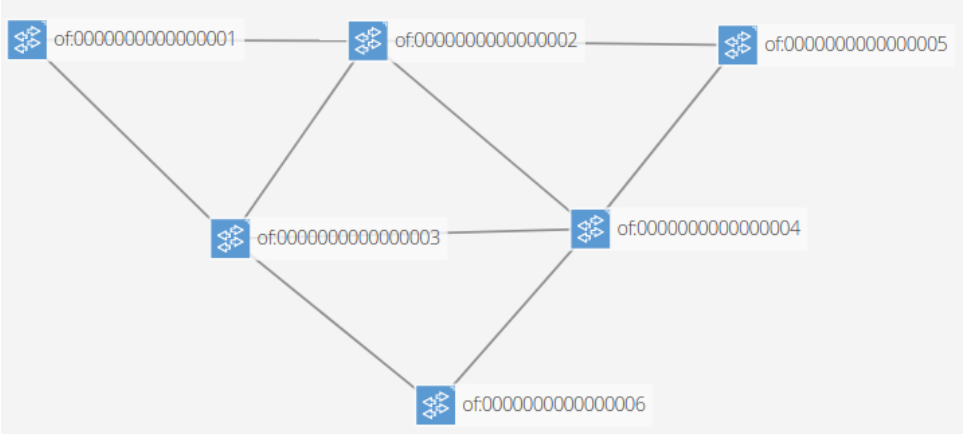
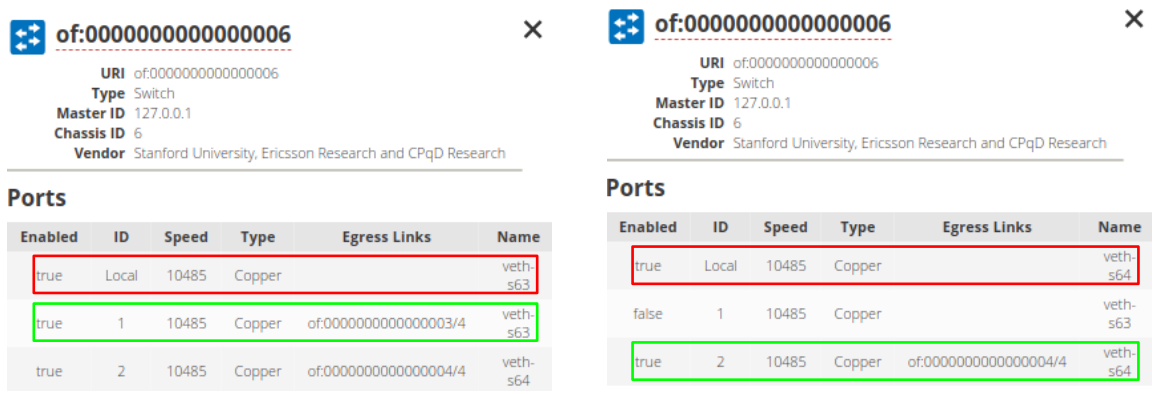


Figura 5.42: Escenario 4 Topología inicial recogida en la interfaz gráfica de ONOS

En primer lugar, se ha utilizado el comando mencionado en el bloque de Código 5.4 para deshabilitar el enlace que comunica el switch 3 y el switch 6, provocando que el switch 6 reconfigure su puerto local dado que inicialmente se había configurado en la interfaz `veth-s63` (Figura 5.43a), que pertenece al enlace que se ha deshabilitado. Examinando el contenido de la tabla de Amaru del switch 6 tras deshabilitar el enlace (Tabla 5.7) se ha observado que los caminos asociados a la interfaz deshabilitada se

han desactivado y que el segundo camino disponible está asociado al puerto 2 del switch, cuya interfaz es `veth-s64` y forma parte del enlace que comunica al switch 6 con el switch 4. Para verificar que el switch 6 ha reconfigurado correctamente su puerto se ha examinado la lista de puertos del switch 6 que muestra ONOS tras deshabilitar el enlace (Figura 5.43b) para confirmar que el puerto local se ha reconfigurado en la interfaz `veth-s64`, coincidiendo con lo deducido anteriormente al examinar la tabla de Amaru.



(a) Estado inicial de los puertos del switch 6

(b) Estado de los puertos del switch 6 tras reconfigurar su puerto local

Figura 5.43: Escenario 4: Evolución de los puertos del switch 6

Siguiendo con la verificación de que los cambios producidos al deshabilitar este primer enlace son correctos, se ha analizado el contenido de la tabla de flujos del switch 4, que se muestra en la Figura 5.44, constatando que el switch 4 ha instalado las reglas necesarias para encaminar el tráfico OpenFlow del switch 6. En la tabla de flujos se puede comprobar que la regla 3 la permite encaminar correctamente los ARPs con destino el switch 6 y las reglas 4 y 5 permiten encaminar correctamente el tráfico OpenFlow entre el switch 6 y el controlador.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="1798", byte_cnt="625276", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="1929", byte_cnt="332820", **insts=[]**},

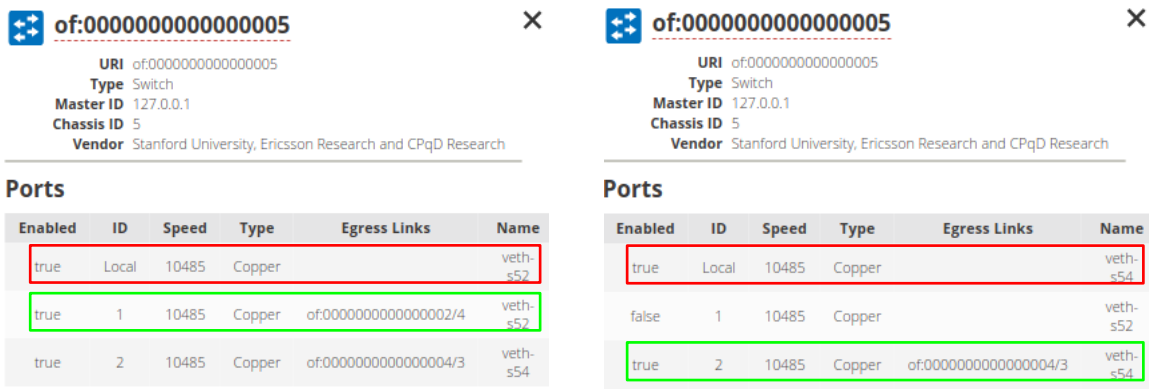
3. match="oxm{**eth_dst="00:00:00:00:00:06", eth_type="0x806"**}", prio="65521", idle_to="10", hard_to="0", pkt_cnt="1", byte_cnt="60", **insts=[apply{acts=[out{port="4"}]}]**},

4. match="oxm{**in_port="4", eth_type="0x800", ipv4_src="10.0.0.106", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="34", byte_cnt="7633", **insts=[apply{acts=[out{port="1"}]}]**},

5. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.106", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="31", byte_cnt="3902", **insts=[apply{acts=[out{port="4"}]}]**},

Figura 5.44: Escenario 4: Conetenido de la tabla de flujos del switch 4 tras la reconfiguración del puerto local del switch 6

En segundo lugar, se ha deshabilitado el enlace que comunica el switch 5 con el switch 2, provocando que el switch 5 reconfigure su puerto local puesto que estaba configurado inicialmente en la interfaz `veth-s52` (Figura 5.45a), que pertenece a este segundo enlace deshabilitado. Seguidamente, se ha examinado la tabla de Amaru del switch 5 (Tabla 5.8) comprobando que los caminos asociados a la interfaz `veth-s52` se han deshabilitado. Asimismo, se ha observado que el siguiente camino disponible es el número 3. Este camino está asociado al puerto 2 y su interfaz `veth-s54` pertenece al enlace que comunica al switch 5 con el switch 4. Para confirmar que el switch 5 ha configurado correctamente su puerto local, se ha revisado la lista de puertos del switch que muestra ONOS tras deshabilitar el segundo enlace (5.45b), donde se ha podido comprobar que el nuevo puerto local se ha configurado en la interfaz esperada, es decir, coincide con el contenido de la tabla de Amaru.



(a) Estado inicial de los puertos del switch 5

(b) Estado de los puertos del switch 5 tras reconfigurar su puerto local

Figura 5.45: Escenario 4: Evolución de los puertos del switch 5

Posición	AMAC	Nivel	Puerto	Activa
1	1:3:4	3	1	0
2	1:2:3:4	4	2	1
3	1:3:3:4	4	2	1
4	1:2:2:4	5	1	0
5	1:2:4:2:4	5	2	1
6	1:3:2:3:4	5	2	1

Tabla 5.7: Escenario 4: Contenido de la tabla de Amaru del switch 6 tras reconfigurar una vez su puerto local

Posición	AMAC	Nivel	Puerto	Activa
1	1:2:4	3	1	0
2	1:3:2:4	4	1	0
3	1:2:3:3	4	2	1
4	1:3:3:1:4	5	1	0
5	1:3:4:2:3	5	2	1
6	1:2:2:3:3	5	2	1

Tabla 5.8: Escenario 4: Contenido de la tabla de Amaru del switch 5 tras reconfigurar una vez su puerto local

Nuevamente, se ha inspeccionado la tabla de flujos del switch 4 para confirmar que también ha instalado los flujos necesarios para encaminar el tráfico OpenFlow del switch 5 (Figura 5.46). Como se ha visto en capturas anteriores, el switch ha instalado las reglas necesarias para encaminar correctamente los paquetes ARP con destino al switch 5, así como los mensajes OpenFlow entre el controlador y el switch 5.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="2407", byte_cnt="832790", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="2586", byte_cnt="443454", **insts=[]**},

3. match="oxm{**in_port="4", eth_type="0x800", ipv4_src="10.0.0.106", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="364", byte_cnt="123489", **insts=[apply{acts=[out{port="1"}]}]**},

4. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.106", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="365", byte_cnt="46886", **insts=[apply{acts=[out{port="4"}]}]**},

5. match="oxm{**eth_dst="00:00:00:00:00:05", eth_type="0x806"**}", prio="65521", idle_to="10", hard_to="0", pkt_cnt="1", byte_cnt="60", **insts=[apply{acts=[out{port="3"}]}]**},

6. match="oxm{**in_port="3", eth_type="0x800", ipv4_src="10.0.0.105", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="7", byte_cnt="466", **insts=[apply{acts=[out{port="1"}]}]**},

7. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.105", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="8", byte_cnt="560", **insts=[apply{acts=[out{port="3"}]}]**},

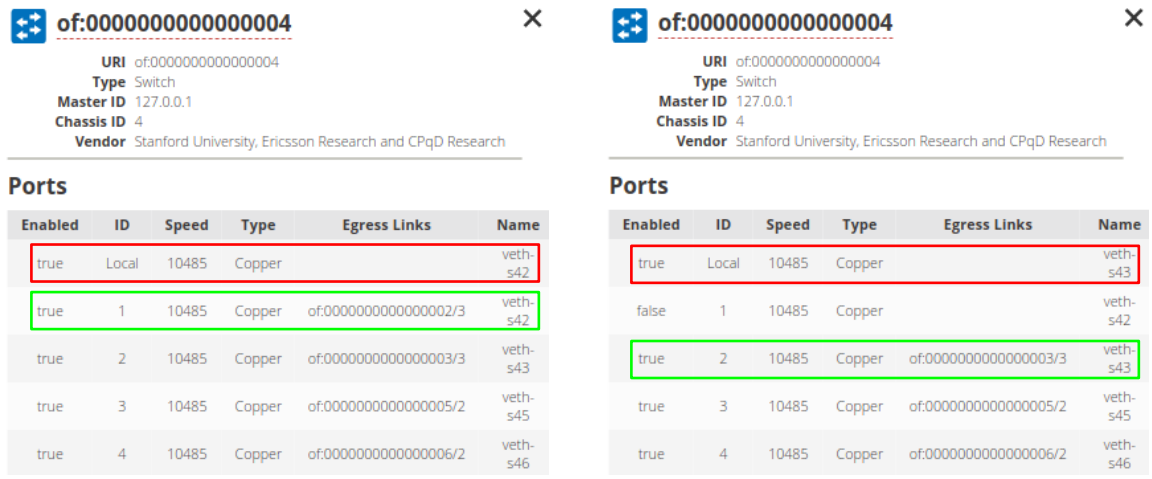
Figura 5.46: Escenario 4: Conetenido de la tabla de flujos del switch 4 tras la reconfiguración del puerto local del switch 5

Por último, se ha deshabilitado un tercer enlace. Este enlace es el que comunica el switch 2 con el switch 4. Al deshabilitarlo, se ha desactivado la interfaz `veth-s42`, que estaba configurada como puerto local en el switch 4 (Figura 5.47a), obligándolo de esta manera a reconfigurar uno nuevo. A continuación, se ha procedido del mismo modo que en los dos casos anteriores, consultando la tabla de Amaru del switch 4 (Tabla 5.9), donde se ha observado que se han deshabilitado los caminos asociados a la interfaz `veth-s42` y que el siguiente camino disponible es el segundo, asociado al puerto 2 cuya interfaz es `veth-s43` y pertenece al enlace que comunica el switch 4 con el switch 3.

Posición	AMAC	Nivel	Puerto	Activa
1	1:2:3	3	1	0
2	1:3:3	3	2	1
3	1:2:4:2	4	3	1
4	1:3:4:2	4	4	1
5	1:3:2:3	4	1	0
6	1:2:2:3	4	2	1

Tabla 5.9: Contenido final de la tabla de Amaru del switch 4

Seguidamente, se ha confirmado que la nueva interfaz configurada como puerto de control coincide con lo deducido previamente a partir del contenido de la tabla de Amaru, mediante la lista de puertos del switch 4 que muestra ONOS después de deshabilitar el tercer enlace (Figura 5.47b). Asimismo, se ha comprobado que el switch 4 ha modificado las entradas de su tabla de flujos para adaptarlas al nuevo puerto local y enciaminar correctamente los flujos OpenFlow de los switchs 5 y 6, que estaba manejando previamente.



(a) Estado inicial de los puertos del switch 4

(b) Estado de los puertos del switch 4 tras reconfigurar su puerto local

Figura 5.47: Escenario 4: Evolución de los puertos del switch 4

Para finalizar la comprobación de los cambios producidos al deshabilitar este tercer enlace, se han consultado las tablas de flujos del switch 4 y del switch 3. En la Figura 5.48 se muestra el contenido de la tabla de flujos del switch 3. En ella se ha resaltado el nuevo puerto de entrada que el switch 4 ha asignado a las instrucciones DROP y el nuevo puerto de salida que ha asignado a las instrucciones que encaminan los flujos TCP con destino al controlador para que coincida con el puerto asociado a la interfaz `veth-s43`, configurada como nuevo puerto local.

1. match="oxm{**in_port="2", eth_src="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="477", byte_cnt="168269", **insts=[]**},

2. match="oxm{**in_port="2", eth_dst="00:00:00:00:00:04"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="471", byte_cnt="77932", **insts=[]**},

3. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.105", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="324", byte_cnt="39444", **insts=[apply{acts=[out{port="3"}]}]**},

4. match="oxm{**in_port="3", eth_type="0x800", ipv4_src="10.0.0.105", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="288", byte_cnt="101626", **insts=[apply{acts=[out{port="2"}]}]**},

5. match="oxm{**in_port="4", eth_type="0x800", ipv4_src="10.0.0.106", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="290", byte_cnt="101576", **insts=[apply{acts=[out{port="2"}]}]**},

6. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.106", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="297", byte_cnt="37726", **insts=[apply{acts=[out{port="4"}]}]**},

Figura 5.48: Escenario 4: Conetenido de la tabla de flujos del switch 4 tras la reconfiguración del puerto local del switch 4

Finalmente, el contenido más relevante de la tabla de flujos del switch 3, después de que el switch 4 reconfigure su puerto local, se muestra en la Figura 5.49. En ella se puede comprobar que el switch 3 ha instalado tanto las reglas necesarias para encaminar el tráfico OpenFlow del switch 4, como las necesarias para encaminar el tráfico del switch 5 y del switch 6. Estos dos últimos flujos OpenFlow eran encaminados previamente por el switch 4 por lo que el switch 3 ha tenido que incluirlos en su tabla de flujos una vez el switch 4 los ha reencaminado a través de su nuevo puerto de control.

1. match="oxm{**in_port="1", eth_src="00:00:00:00:00:03"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="3257", byte_cnt="1181644", **insts=[]**},

2. match="oxm{**in_port="1", eth_dst="00:00:00:00:00:03"**}", prio="65535", idle_to="0", hard_to="0", pkt_cnt="3553", byte_cnt="711208", **insts=[]**},

3. match="oxm{**eth_dst="00:00:00:00:00:04", eth_type="0x806"**}", prio="65521", idle_to="10", hard_to="0", pkt_cnt="1", byte_cnt="60", **insts=[apply{acts=[out{port="3"}]}]**},

4. match="oxm{**in_port="3", eth_type="0x800", ipv4_src="10.0.0.104", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="475", byte_cnt="167444", **insts=[apply{acts=[out{port="1"}]}]**},

5. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.104", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="470", byte_cnt="77872", **insts=[apply{acts=[out{port="3"}]}]**},

6. match="oxm{**in_port="3", eth_type="0x800", ipv4_src="10.0.0.105", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="288", byte_cnt="101626", **insts=[apply{acts=[out{port="1"}]}]**},

7. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.105", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="324", byte_cnt="39444", **insts=[apply{acts=[out{port="3"}]}]**},

8. match="oxm{**in_port="3", eth_type="0x800", ipv4_src="10.0.0.106", ipv4_dst="10.0.0.88", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="291", byte_cnt="101872", **insts=[apply{acts=[out{port="1"}]}]**},

9. match="oxm{**eth_type="0x800", ipv4_src="10.0.0.88", ipv4_dst="10.0.0.106", ip_proto="6"**}", prio="65521", idle_to="5", hard_to="0", pkt_cnt="297", byte_cnt="37726", **insts=[apply{acts=[out{port="3"}]}]**},

Figura 5.49: Escenario 4: Conetenido de la tabla de flujos del switch 3 tras la reconfiguración del puerto local del switch 4

Mediante las pruebas que se acaban de realizar, se ha comprobado que los distintos switches desplegados en el entorno son capaces de reconfigurar correctamente sus puertos de control utilizando los caminos generados por Amaru y su lógica *in-band* consigue adaptarse a los cambios que se producen en la red.

Tras concluir las pruebas del escenario 4, se ha capturado en la Figura 5.50 la topología resultante en la interfaz gráfica de ONOS, tras haber deshabilitado los 3 enlaces. En ella se puede comprobar que el switch 6 ha dejado de estar directamente comunicado con el switch 3 y que los switches 4 y 5 ya no tienen un enlace directo con el switch 2.

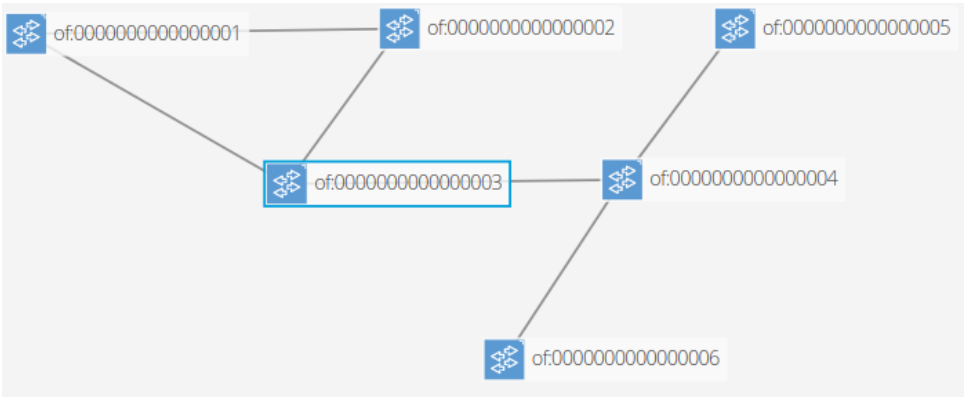


Figura 5.50: Escenario 4 Topología final recogida en la interfaz gráfica de ONOS

5.4 Conclusiones

Una vez concluido el conjunto de pruebas establecido para los distintos escenarios se han extraído varias conclusiones de los resultados obtenidos:

- La implementación del control *in-band* funciona de la manera prevista y permite establecer realmente este tipo de control si los switches BOFUSS de la topología se encapsulan en un `network namespace` y sus interfaces se configuran adecuadamente.
- La implementación de la reconfiguración del puerto de control mediante los caminos generados por Amaru funciona correctamente y permite mantener los flujos de los dispositivos OpenFlow que pudiera manejar anteriormente, dado que es capaz de modificar las entradas de los flujos para adaptarlas al nuevo puerto de control.
- La reconfiguración del puerto de control se realiza en base a los caminos disponibles de la tabla de Amaru.
- Las reglas instaladas inicialmente cuando está activado el modo de control *in-band*, que se utilizan para enviar el tráfico ARP y TCP al controlador con el objetivo de poder encaminar el tráfico OpenFlow de otros dispositivos, no sería necesarias con ninguno de los dos controladores utilizados, dado que ambos instalan reglas que pueden sustituirlas. No obstante, se ha decidido mantenerlas ya que son representativas a la hora de identificar el tipo de tráfico que es necesario encaminar para permitir la conexión de nuevos dispositivos OpenFlow.
- Ambos controladores son muy útiles, aunque, a pesar de tener una instalación más tediosa, ONOS resulta mucho más cómodo de utilizar y más estable a la hora de emplear redes con un gran número de dispositivos.
- En los escenarios en los que se han desplegado `hosts` se ha comprobado que mantienen la comunicación tras la reconfiguración de los puertos de control de los switches del entorno.

Capítulo 6

Conclusiones y líneas futuras

A lo largo de este capítulo se exponen las conclusiones obtenidas una vez finalizado el proyecto y se proponen futuras líneas de desarrollo y mejora del trabajo realizado.

6.1 Conclusiones

En este proyecto se ha llevado a cabo una primera implementación que permite la reconfiguración del puerto de control de un switch BOFUSS haciendo uso de los caminos generados por el protocolo Amaru. Gracias a las modificaciones que incorpora esta implementación, se ha logrado proporcionar una solución que encamina de manera *in-band* el tráfico de control. Para alcanzar este cometido, ha sido necesario estudiar y comprender las tecnologías, protocolos y herramientas que enmarcan este trabajo. En primer lugar, ha sido indispensable comprender la aquitectura de la tecnología de las Redes Definidas por *Software* para obtener una visión clara acerca del tipo de dispositivo que iba a emplearse para llevar a cabo las implementaciones, así como de los elementos con los que debe interactuar. A continuación, se ha realizado un breve análisis de las herramientas y dispositivos propuestos para finalmente elegir los utilizados durante el desarrollo del proyecto. Una vez elegidas las distintas herramientas, se ha estudiado el protocolo Amaru para comprender su funcionamiento y poder concretar las funcionalidades relacionadas con el protocolo que deberán ser implementadas posteriormente en el dispositivo switch.

Después de haber comprendido el protocolo Amaru, se ha llevado a cabo un análisis exhaustivo de la estructura del switch BOFUSS y de su funcionamiento para lograr acotar las modificaciones necesarias para satisfacer los objetivos propuestos. Posteriormente, ha sido preciso modificar el switch BOFUSS para lograr que el modo de control *in-band* funcionara correctamente para, seguidamente, implementar la reconfiguración del puerto de control mediante Amaru. Esta reconfiguración se hace en base al puerto que tiene asociado cada camino o entrada de la tabla Amaru, y no haciendo uso de las AMACs. Esto es debido a que no existe la necesidad de diferenciar entre diferentes nodos raíz, ya que únicamente se ha utilizado uno y, por consiguiente, existe un único árbol. Para finalizar, se han llevado a cabo varias pruebas en distintos escenarios para asegurar el correcto funcionamiento de las nuevas implementaciones. Durante el desarrollo de las pruebas se ha verificado que el switch mantiene las funcionalidades que ofrecía antes de incorporar las modificaciones, además de confirmar que el switch BOFUSS es capaz de funcionar correctamente mediante el modo de control *in-band* cuando el entorno está configurado correctamente. Por último, se ha comprobado que es capaz de reconfigurar el puerto local haciendo uso de los caminos generados por el protocolo Amaru.

A modo de resumen, en este proyecto se han realizado las siguiente contribuciones:

- Implementación del modo de control *in-band* en el switch BOFUSS.
- Implementación de la reconfiguración del puerto de control del switch BOFUSS mediante la información de los caminos generada por el protocolo Amaru.
- Desarrollo de una solución que permite encaminar el tráfico de control *in-band* en entornos SDN y es capaz de recuperarse ante la caída de enlaces a través de los que también se transmite información de control de los switches BOFUSS.
- Creación de un conjunto de escenarios que permiten verificar si las nuevas funcionalidades implementadas en el switch BOFUSS funcionan correctamente.

6.2 Líneas futuras

Tras identificar los objetivos cumplidos y las aportaciones logradas mediante la realización de este proyecto en el apartado anterior, en esta sección se describen posibles mejoras y líneas futuras de desarrollo:

- Una vez se ha obtenido una versión estable del switch BOFUSS que incorpore las nuevas funcionalidades, sería interesante diseñar un conjunto de pruebas en el que se compare el rendimiento de este switch con el switch BOFUSS original.
- Implementar un mecanismo que permita que el switch, en el momento de reconfigurar un nuevo puerto de control, sea capaz de interpretar como un puerto inválido aquel por el que recibe mensajes OpenFlow de otros switches. De esta manera se pretende evitar que se produzcan bucles y que los dispositivos cuyas conexiones OpenFlow cursen a través de dicho switch pierdan la comunicación con el controlador.
- Recopilar información del funcionamiento del switch con otros controladores para definir las reglas que se instalan inicialmente para utilizar el modo de control *in-band*, de tal modo que sea compatible con la mayoría de controladores.
- Adaptar la implementación para utilizar dos o más switches raíz (*root*), es decir, que la red disponga de dos o más puntos de conexión con el controlador y se generen varios árboles diferentes.
- Automatizar la configuración inicial del puerto local del switch para evitar tener que configurar inicialmente los entornos.
- Añadir seguridad mediante el cifrado de las conexiones OpenFlow *in-band* puesto que, hasta ahora, se han establecido utilizando TCP sin cifrar.

Bibliografía

[1] A. Jalili, H. Nazari, S. Namvarasl, and M. Keshtgari, "A comprehensive analysis on control plane deployment in SDN: In-band versus out-of-band solutions," in *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*. IEEE, 2017, pp. 1025–1031.

[2] Open Networking Foundation, "SDN Architecture," pp. 1–68, 2014. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf

[3] ——, "SDN Architecture Overview," 2013. [Online]. Available: https://opennetworking.org/wp-content/uploads/2013/02/SDN-architecture-overview-1.0.pdf

[4] "OpenFlow Switch Specification 1.3." [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf

[5] E. L. Fernandes, E. Rojas, J. Alvarez-Horcajo, Z. L. Kis, D. Sanvito, N. Bonelli, C. Cascone, and C. E. Rothenberg, "The road to BOFUSS: The basic OpenFlow userspace software switch," *Journal of Network and Computer Applications*, vol. 165, p. 102685, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804520301594

[6] E. L. Fernandes, "Software Switch 1.3: An experimenter-friendly OpenFlow implementation," Ph.D. dissertation, UNIVERSIDADE ESTADUAL DE CAMPINAS, 2015. [Online]. Available: https://www.dca.fee.unicamp.br/~chesteve/thesis/Dissertacao-Eder-SoftSwitch13-20150416.pdf

[7] "The introduction to OVS architecture." [Online]. Available: https://hustcat.github.io/an-introduction-to-ovs-architecture/

[8] D. Lopez-Pajares, J. Alvarez-Horcajo, E. Rojas, A. S. M. Asadujjaman, and I. Martinez-Yelmo, "Amaru: Plug play resilient in-band control for sdn," *IEEE Access*, vol. 7, pp. 123 202–123 218, 2019.

[9] "ONOS Tutorial | SDN Hub." [Online]. Available: http://sdnhub.org/tutorials/onos/

[10] "Ryu sdn framework." [Online]. Available: https://www.slideshare.net/yamahata/ryu-sdnframeworkupload

[11] "Install Ryu Controller (Ubuntu 14.04.3 Server) · GitHub." [Online]. Available: https://gist.github.com/olegslavkin/e01ccc1835396402dc2f

[12] "Redes definidas por software (SDN) - Descripción general - Cisco." [Online]. Available: https://www.cisco.com/c/es_cr/solutions/software-defined-networking/overview.html

[13] "Se disparan las expectativas sobre el mercado de sdn," https://www.networkworld.es/networking/se-disparan-las-expectativas-sobre-el-mercado-de-sdn [Útimo acceso 20/noviembre/2018].

[14] "Importante crecimiento de las redes sdn," https://www.networkworld.es/networking/las-sdn-experimentaran-un-importante-crecimiento-en-los-proximos-cuatro-anos [Útimo acceso 20/noviembre/2018].

[15] "CPqD/ofsoftswitch13: OpenFlow 1.3 switch." [Online]. Available: https://github.com/CPqD/ofsoftswitch13

[16] "Open vSwitch." [Online]. Available: https://www.openvswitch.org/

[17] M. Lessing and C. Craven, *SDN 101 Network Foundations Guide*. SDxCentral LLC, 2019.

[18] W. Stallings, "Software-Defined Networks," *The Internet Protocol Journal*, vol. 16, no. 1, March 2013.

[19] Open Networking Foundation, "SDN Architecture issue 1.1." [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/TR-521_SDN_Architecture_issue_1.1.pdf

[20] "What is OpenFlow? Definition and How it Relates to SDN." [Online]. Available: https://www.sdxcentral.com/networking/sdn/definitions/what-is-openflow/

[21] "OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06)." [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[22] "TrafficLab/of11softswitch: OpenFlow 1.1 Softswitch." [Online]. Available: https://github.com/TrafficLab/of11softswitch

[23] "Netbee." [Online]. Available: https://github.com/netgroup-polito/netbee

[24] F. Risso and M. Baldi, "NetPDL: An extensible XML-based language for packet header description," *Computer Networks*, vol. 50, no. 5, pp. 688–706, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128605002008

[25] "Software-Defined Networking (SDN) with OpenStack." [Online]. Available: https://www.packtpub.com/product/software-defined-networking-sdn-with-openstack/9781786465993

[26] "ONOS - ONOS - Wiki." [Online]. Available: https://wiki.onosproject.org/display/ONOS/ONOS

[27] "Ryu SDN Framework." [Online]. Available: https://ryu-sdn.org/

[28] "Ryu application API Ryu 4.34 documentation." [Online]. Available: https://ryu.readthedocs.io/en/latest/ryu_app_api.html

[29] "ip-netns(8) - Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man8/ip-netns.8.html

[30] "veth(4) - Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man4/veth.4.html

[31] "Implementación de Amaru en el switch BOFUSS." [Online]. Available: https://github.com/gistnetserv-uah/Amaru

[32] "Ubuntu Manpage: tap - Ethernet tunnel software network interface." [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/en/man4/tap.4freebsd.html

[33] "Visual Studio Code - Code Editing. Redefined." [Online]. Available: https://code.visualstudio.com/

[34] "C++ programming with Visual Studio Code." [Online]. Available: https://code.visualstudio.com/docs/languages/cpp

[35] "GDB: The GNU Project Debugger." [Online]. Available: https://www.gnu.org/software/gdb/

[36] "Ubuntu 16.04.7 LTS (Xenial Xerus)." [Online]. Available: https://releases.ubuntu.com/16.04/

[37] "Introduction to TCP Offload Engines." [Online]. Available: https://www.dell.com/downloads/global/power/1q04-her.pdf

[38] "ethtool(8) - Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man8/ethtool.8.html

[39] "ping(8) - Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man8/ping.8.html

[40] "Logo de ONOS utilizado en las capturas." [Online]. Available: https://opennetworking.org/onos/

[41] "Icono utilizado como controlador en las capturas." [Online]. Available: https://www.flaticon.es/icono-gratis/codigo_2920277?related_id=2920277&origin=pack

Anexo A

Manuales de instalación

Los siguientes manuales han sido probados en un equipo cuyo sistema operativo es **Ubuntu Desktop 16.04 LTS** y cuya versión de ***kernel*** es **4.15.0-132-generic**.

A.1 Manual de instalación del software switch BOFUSS

Para instalar el switch BOFUSS se ha utilizado el siguiente *script*:

Código A.1: Script de instalación del switch BOFUSS

```
#!/bin/sh
echo "Se instalan los prerrequisitos"
sudo apt-get install cmake libpcap-dev libxerces-c3.1 libxerces-c-dev libpcre3
    libpcre3-dev flex bison pkg-config autoconf libtool libboost-dev

echo "Se instalan la libreria netbee"
git clone https://github.com/netgroup-polito/netbee.git
cd netbee/src
cmake .
make
sudo cp ../bin/libn*.so /usr/local/lib
sudo ldconfig
sudo cp -R ../include/* /usr/include/

echo "Se instala el ofsoftswitch13"
cd ~/
git clone https://github.com/CPqD/ofsoftswitch13.git
cd ofsoftswitch13
./boot.sh
./configure
make
sudo make install
```

A.2 Manual de instalación del controlador ONOS

Para instalar el controlador ONOS se ha utilizado el siguiente *script*:

Código A.2: Script de instalación del controlador ONOS

```
#!/bin/sh
echo "Se instala la herramienta BAZEL y sus prerrequisitos"
sudo apt install g++ unzip zip
sudo apt-get install openjdk-8-jdk

#Se instala BAZEL
wget https://github.com/bazelbuild/bazel/releases/download/1.2.1/bazel-1.2.1-
    installer-linux-x86_64.sh
chmod +x bazel-1.2.1-installer-linux-x86_64.sh
./bazel-1.2.1-installer-linux-x86_64.sh --user
echo "#Bazel" | tee -a ~/.bashrc
echo "export PATH="\$PATH:\$HOME/bin"" | tee -a ~/.bashrc
#Se instala KARAF
cd ~
mkdir Applications
wget http://archive.apache.org/dist/karaf/4.2.4/apache-karaf-4.2.4.tar.gz
tar -zxvf apache-karaf-4.2.4.tar.gz -C ~/Applications/

echo "Se instala ONOS-2.2"
cd ~
git clone https://gerrit.onosproject.org/onos -b onos-2.2
echo "#ONOS" | tee -a ~/.bashrc
echo ". ~/onos/tools/dev/bash_profile" | tee -a ~/.bashrc
. ~/.bashrc
cd ~/onos
bazel build onos
#Para iniciar ONOS:
#bazel run onos-local

echo "Para iniciar el controlador ONOS se utiliza <bazel run onos-local>"
```

A.3 Manual de instalación del controlador RYU

Para instalar el controlador RYU se ha utilizado el siguiente *script*:

Código A.3: Script de instalación del controlador RYU [11]

```
#!/bin/sh
echo "Se instalan los prerrequisitos"
sudo apt-get -y install git python-pip python-dev
sudo apt-get -y install python-eventlet python-routes python-webob python-
    paramiko

echo "Se instala el controlador RYU"
cd ~/
git clone git://github.com/osrg/ryu.git
sudo pip install setuptools --upgrade
cd ryu;
sudo python ./setup.py install

echo "Se instalan y actualizan los paquetes python"
sudo pip install tinyrpc
sudo pip install six --upgrade
sudo pip install oslo.config msgpack-python
sudo pip install eventlet --upgrade
```

Anexo B

Código generado para programar las nuevas implementaciones

B.1 Implementación puerto local

Código B.1: Modificaciones en la función port_watcher_local_packet_cb()

```
if (ntohs(repl->type) == OFPMP_PORT_DESC)
        {
            bool seen[PORT_ARRAY_SIZE];
            struct ofp_port *p;
            unsigned int port_no;
            size_t n_ports, i;
            p = (struct ofp_port *)repl->body;
            memset(seen, false, sizeof seen);
            n_ports = ((msg->size - offsetof(struct ofp_multipart_reply, body)) /
    sizeof *p);
            for (i = 0; i < n_ports; i++)
            {
                struct ofp_port *opp = &p[i];
                //***Modificaciones Boby UAH***//
                port_no = (ntohl(opp->port_no) == OFPP_LOCAL) ? OFPP_LOCAL_AUX :
    ntohl(opp->port_no);
                /* if (ntohl(opp->port_no) > PORT_ARRAY_SIZE - 1) {*/
                if (port_no > PORT_ARRAY_SIZE - 1)
                {

                    if (port_no <= OFPP_MAX)
                    {
                        VLOG_WARN(LOG_MODULE, "Port ID %u over limit", ntohl(opp->
    port_no));
                    }
                    continue;
                }
                /*++++FIN++++/
```

```
                update_phy_port(pw, opp, OFPPR_MODIFY);
                /*Modificaciones Boby UAH*/
                /* seen[ntohl(opp->port_no)] = true;*/
                seen[port_no] = true;
                /*+++FIN+++*/
            }

...
```

Código B.2: Modificaciones en la función update_phy_port()

```
\UseRawInputEncoding
update_phy_port(struct port_watcher *pw, struct ofp_port *opp,
                uint8_t reason)
{
    struct ofp_port *old;
    uint32_t port_no;
    //**Modificaciones Boby UAH**//
    /*port_no = ntohl(opp->port_no);*/
    port_no = (ntohl(opp->port_no) == OFPP_LOCAL) ? OFPP_LOCAL_AUX : ntohl(opp->
    port_no);
    /*+++FIN+++*/

    old = lookup_port(pw, port_no);

...
```

Código B.3: Modificaciones en la función new_port() de dp_ports.c

```
...
//+++Modificaciones Boby UAH+++//
// Se configura el bit OFPPC_NO_FWD para que no se use el puerto local con
    OFPP_FLOOD
if (port_no == OFPP_LOCAL)
{
    port->conf->config |= OFPPC_NO_FWD;
}
//++++++//

...
```

Código B.4: Modificaciones en la función dp_ports_output_all() de dp_ports.c

```
int dp_ports_output_all(struct datapath *dp, struct ofpbuf *buffer, int in_port,
    bool flood)
{
    struct sw_port *p;
    //+++Modificaciones Boby UAH+++//
    LIST_FOR_EACH(p, struct sw_port, node, &dp->port_list)
    {
        if (p->conf->port_no == in_port)
        {
            continue;
```

```
        }
        if (in_port == OFPP_LOCAL && !strcmp(p->conf->name, dp->local_port->conf->
    name)) // Se comparan los nombres de los puertos porque el el puerto local y
    otro puerto fÃsico comparten la misma interfaz)
        {
            continue;
        }
        if (flood && (p->conf->config & OFPPC_NO_FWD))
        {
            continue;
        }
        dp_ports_output(dp, buffer, p->conf->port_no, 0);
    }
    return 0;
}
```

B.2 Implementación del modo in-band

Código B.5: Modificaciones en la función make_flow_mod()

```
struct ofpbuf *
make_flow_mod(uint8_t command, uint8_t table_id,
              const struct flow *flow, size_t actions_len)
{
    struct ofp_flow_mod *ofm;
    size_t size;
    struct ofpbuf *out;

    //**Modificaciones UAH**//

    uint8_t *dir_m;
    struct ofl_match *ofl_m = xmalloc(sizeof(struct ofl_match));
    struct flow flow_aux = *flow;
    ofl_structs_match_init(ofl_m);
    create_ofl_match_UAH(&flow_aux, ofl_m); //Se invoca a la funcion para crear el
    ofl_match

    size = ROUND_UP(sizeof(struct ofp_flow_mod) - 4 + ofl_m->header.length, 8) +
    actions_len; //ofl_m->header.length representa el tamano de los campos de match
    out = ofpbuf_new(size);
    ofm = ofpbuf_put_zeros(out, sizeof *ofm);
    dir_m = (uint8_t *)ofm + sizeof(struct ofp_flow_mod) - 4;
          //Direccion de los oxm_fields del campo match
    ofl_structs_match_pack((struct ofl_match_header *)ofl_m, &(ofm->match), dir_m,
    NULL); //Se empaqueta la estructura ofl_match en el ofp_flow_mod
    out->size = ROUND_UP(sizeof(struct ofp_flow_mod) - 4 + ofl_m->header.length, 8)
    ;

    ofm->header.version = OFP_VERSION;
    ofm->header.type = OFPT_FLOW_MOD;
```

```
    ofm->header.length = htons(size);
    ofm->cookie = 0;
    ofm->out_group = OFPG_ANY;
    ofm->out_port = OFPP_ANY;
    ofm->command = command;
    ofm->table_id = table_id;

    ofl_structs_free_match((struct ofl_match_header *)ofl_m, NULL);
    /*+++FIN+++*/

    return out;
}
```

Código B.6: Función create_ofl_match_UAH()

```
static void create_ofl_match_UAH(struct flow *flow, struct ofl_match *match)
{
    uint32_t ip4_aux;
    char ip4[INET_ADDRSTRLEN];

    ofl_structs_match_put16(match, OXM_OF_ETH_TYPE, ntohs(flow->dl_type));
    if (flow->dl_type == htons(ETH_TYPE_ARP))
    {
        if (flow->in_port)
        {
            ofl_structs_match_put32(match, OXM_OF_IN_PORT, ntohl(flow->in_port));
        }
        if (flow->nw_dst)
        {
            ip4_aux = flow->nw_dst;
            inet_ntop(AF_INET, &ip4_aux, ip4, INET_ADDRSTRLEN);
            VLOG_WARN(LOG_MODULE, "[CREATE OFL MATCH UAH]: ARP destino: %u==%s",
    flow->nw_dst, ip4);
            ofl_structs_match_put32(match, OXM_OF_ARP_TPA, flow->nw_dst); //IP
    Target
        }
        if (flow->dl_dst && !eth_addr_is_zero(flow->dl_dst))
        {
            ofl_structs_match_put_eth(match, OXM_OF_ETH_DST, flow->dl_dst);
        }
    }
    else if (flow->dl_type == htons(ETH_TYPE_IP) && flow->nw_proto == IP_TYPE_TCP)
    {

        ofl_structs_match_put8(match, OXM_OF_IP_PROTO, flow->nw_proto); // TCP
        if (flow->in_port)
        {
            ofl_structs_match_put32(match, OXM_OF_IN_PORT, ntohl(flow->in_port));
        }
        if (flow->nw_dst)
        {
```

```
            ofl_structs_match_put32(match, OXM_OF_IPV4_DST, flow->nw_dst); // IP
    Destino (Controlador)
        }
        if (flow->nw_src)
        {
            ofl_structs_match_put32(match, OXM_OF_IPV4_SRC, flow->nw_src); // IP
    Destino (Controlador)
        }
        if (flow->tp_dst)
        {
            ofl_structs_match_put16(match, OXM_OF_TCP_DST, ntohs(flow->tp_dst)); //
     Puerto de la conexion OpenFlow
        }
    }
}
```

Código B.7: Modificaciones en la función make_add_flow()

```
struct ofpbuf *
make_add_flow(const struct flow *flow, uint32_t buffer_id, uint8_t table_id,
              uint16_t idle_timeout, size_t actions_len, uint16_t priority)
{
    //**Modificaciones Boby UAH**//
    struct ofp_flow_mod *ofm;
    struct ofpbuf *out;
    //++++++//
    // Se configura un drop para este flujo
    if (actions_len == 0)
    {
        out = make_flow_mod(OFPFC_ADD, table_id, flow, 0);
    }
    else
    {
        struct ofp_instruction_actions *oia;
        size_t instruction_len = sizeof *oia + actions_len;
        out = make_flow_mod(OFPFC_ADD, table_id, flow, instruction_len);
        /* Use a single apply-actions for now - Jean II */
        oia = ofpbuf_put_zeros(out, sizeof *oia);
        oia->type = htons(OFPIT_APPLY_ACTIONS);
        oia->len = htons(instruction_len);
    }

    ofm = out->data;
    // Se configuran asÃ los timeouts para que las reglas sean permanentes
    ofm->idle_timeout = htons(idle_timeout);
    ofm->hard_timeout = htons(OFP_FLOW_PERMANENT);
    ofm->buffer_id = htonl(buffer_id);
    // Se configura la prioridad de la regla.
    ofm->priority = htons(priority);

    return out;
```

```
}
```

Código B.8: Modificaciones en la función make_add_simple_flow()

```
struct ofpbuf *
make_add_simple_flow(const struct flow *flow,
                     uint32_t buffer_id, uint32_t out_port,
                     uint16_t idle_timeout, uint16_t priority)
{
    struct ofpbuf *buffer;
    /*Modificaciones Boby UAH*/
    struct ofl_msg_header *ofl_oh;
    /*+++FIN+++*/

    // Con out_port = 0 se especifica que es un drop
    if (out_port != 0)
    {
        struct ofp_action_output *oao;
        buffer = make_add_flow(flow, buffer_id, 0x00, idle_timeout, sizeof *oao,
    priority);
        oao = ofpbuf_put_zeros(buffer, sizeof *oao);
        oao->type = htons(OFPAT_OUTPUT);
        oao->len = htons(sizeof *oao);
        oao->port = htonl(out_port);
        oao->max_len = OFPCML_NO_BUFFER; //Para que envÃe el paquete completo en
    el packet_in

        VLOG_WARN(LOG_MODULE, "[MAKE ADD SIMPLE FLOW]: FLOW MOD NORMAL!");
    }
    else
    {
        buffer = make_add_flow(flow, buffer_id, 0, idle_timeout, 0, priority);


        VLOG_WARN(LOG_MODULE, "[MAKE ADD SIMPLE FLOW]: FLOW MOD DROP!");
    }


    if (!ofl_msg_unpack(buffer->data, buffer->size, &ofl_oh, NULL /*xid*/, NULL))
    {
        return buffer;
    }
    else
    {
        return NULL;
    }
}
```

Código B.9: Modificaciones en la función make_packet_out()

```
struct ofpbuf *
make_packet_out(const struct ofpbuf *packet, uint32_t buffer_id,
```

```
                uint32_t in_port,
                const struct ofp_action_header *actions, size_t n_actions)
{
    //Modificaciones UAH//
    size_t actions_len = n_actions * ntohs(actions->len);
    // size_t actions_len = n_actions * sizeof *actions;
    //****//

    struct ofp_packet_out *opo;
    size_t size = sizeof *opo + actions_len + (packet ? packet->size : 0);
    struct ofpbuf *out = ofpbuf_new(size);
    opo = ofpbuf_put_uninit(out, sizeof *opo);
    opo->header.version = OFP_VERSION;
    opo->header.type = OFPT_PACKET_OUT;
    opo->header.length = htons(size);
    opo->header.xid = htonl(0);
    opo->buffer_id = htonl(buffer_id);
    opo->in_port = htonl(in_port);
    opo->actions_len = htons(actions_len);
    ofpbuf_put(out, actions, actions_len);
    if (packet)
    {
        ofpbuf_put(out, packet->data, packet->size);
    }
    return out;
}
```

Código B.10: Función get_ofp_packet_eth_header_UAH()

```
bool get_ofp_packet_eth_header_UAH(struct relay *r, struct ofl_msg_packet_in **opip
    , struct eth_header **ethp , uint32_t *in_port, struct ofpbuf **buf)
{
    const int min_len = 0;
    struct ofpbuf b;
    struct eth_header *eth;
    struct ofl_msg_packet_in *oflpi = get_ofl_packet_in_UAH(r, in_port, buf);
    if (oflpi && ntohs(oflpi->data_length) > min_len)
    { // Es data_length porque nos interesa el contenido del packet_in
        *opip = oflpi;
        b.data = (*opip)->data;
        b.size = (*opip)->data_length;

        eth = ofpbuf_try_pull(&b, ETH_HEADER_LEN);
        *ethp = eth;
        return true;
    }

    return false;
}
```

Código B.11: Función get_ofl_packet_in_UAH()

```
struct ofl_msg_packet_in *get_ofl_packet_in_UAH(struct relay *r, uint32_t *in_port,
     struct ofpbuf **buf)
{
    struct ofpbuf *msg = r->halves[HALF_LOCAL].rxbuf;
    struct ofl_msg_header *ofl_oh;
    struct ofp_header *ofp_oh;
    struct ofl_msg_packet_in *ofl_packet_in_p;
    struct ofl_match_tlv *input;

    ofp_oh = (struct ofp_header *)msg->data;

    if ((!ofl_msg_unpack(msg->data, ntohs(ofp_oh->length), &ofl_oh, NULL /*xid*/,
    NULL)) && ofl_oh->type == OFPT_PACKET_IN)
    {
        ofl_packet_in_p = (struct ofl_msg_packet_in *)ofl_oh;
        input = oxm_match_lookup(OXM_OF_IN_PORT, (struct ofl_match *)
    ofl_packet_in_p->match); //InPort match
        *in_port = (uint32_t) * (input->value);
              //Se asigna el valor del in_port extraÃdo del match
        *buf = msg;

        return ofl_packet_in_p;
    }
        return NULL;

}
```

Código B.12: Modificaciones en la función main() de secchan.c

```
int main(int argc, char *argv[])
{
    ...

    while (s.discovery || rconn_is_alive(remote_rconn))
    {
        ...
        if (s.discovery)
        {
            char *controller_name;
            if (rconn_is_connectivity_questionable(remote_rconn))
            {
                discovery_question_connectivity(discovery);
            }
            if (discovery_run(discovery, &controller_name))
            {
                if (controller_name)
                {
                    rconn_connect(remote_rconn, controller_name);
                }
                else
                {
```

```
                    rconn_disconnect(remote_rconn);
                }
            }
        }
        //Modificaciones Boby UAH//
        if (!in_band_rules && rconn_is_connected(local_rconn) &&
    get_pw_local_port_number_UAH(pw))
        {
            install_in_band_rules_UAH(local_rconn, secchan.hooks[2].aux);
            in_band_rules = true;
        }
        /*+++FIN+++*/
        /* Wait for something to happen. */
        LIST_FOR_EACH(r, struct relay, node, &relays)
        {
            relay_wait(r);
        }
        for (i = 0; i < n_listeners; i++)
        {
            pvconn_wait(listeners[i]);
        }
        if (monitor)
        {
            pvconn_wait(monitor);
        }
        for (i = 0; i < secchan.n_hooks; i++)
        {
            if (secchan.hooks[i].class->wait_cb)
            {
                secchan.hooks[i].class->wait_cb(secchan.hooks[i].aux);
            }
        }
        if (discovery)
        {
            discovery_wait(discovery);
        }
        poll_block();
    }

    return 0;
}
```

Código B.13: Modificación de la estructura in_band_data de in_band.c

```
struct in_band_data
{
    const struct settings *s;
    struct mac_learning *ml;
    struct netdev *of_device;
    struct rconn *controller;
    int n_queued;
```

```
    //Modificaciones Boby UAH//
    struct port_watcher *pw; // Para poder buscar el numero del puerto fisico
    compartido con el puerto local
    //+++FIN+++//
};
```

Código B.14: Función get_pw_local_port_number_UAH() de port-watcher.c

```
uint32_t
get_pw_local_port_number_UAH(struct port_watcher *pw)
{
    struct ofp_port *p;
    unsigned int port_no;

    for (p = port_array_first(&pw->ports, &port_no); p;
         p = port_array_next(&pw->ports, &port_no))
    {
        const char *name = (const char *)p->name;
        if (!strcmp(pw->local_port_name, name))
        {
            return port_no;
        }
    }
    return 0;
}
```

Código B.15: Función get_of_port_UAH() en secchan.c

```
uint16_t get_of_port_UAH(const char *controller_str)
{
    char *str = NULL, *str_aux = NULL, *controller_str_aux = NULL;
    int i;
    uint16_t connection_port = 0;
    controller_str_aux = xstrdup(controller_str);

    //controller_str contiene un string del estilo tcp:10.0.0.88:6653 por lo que se
     hace un split del string y
    //se guarda el ultimo bloque que es el puerto de la conexion OpenFlow
    for (str = strtok_r(controller_str_aux, ":", &str_aux), i = 0;
         str != NULL;
         str = strtok_r(NULL, ":", &str_aux), ++i)
    {
        if (i == 2)
        {
            connection_port = atoi(str);
        }
    }
    return connection_port;

}
```

Código B.16: Función install_in_band_rules_UAH() de in_band.c

```
void install_in_band_rules_UAH(struct rconn *local_rconn, struct in_band_data *
    in_band, uint16_t of_port)
{
    struct in_addr local_ip;
    struct flow arp_flow = {0}, tcp_flow = {0}, local_mac_flow = {0};
    uint32_t buffer_id = 0xffffffff, local_port_no;
    arp_flow.dl_type = htons(ETH_TYPE_ARP);
    tcp_flow.dl_type = htons(ETH_TYPE_IP);
    tcp_flow.nw_proto = IP_TYPE_TCP;
    tcp_flow.tp_dst = htons(of_port); //Puerto de la conexion OpenFlow

    rconn_send(local_rconn, make_add_simple_flow(&arp_flow, buffer_id,
    OFPP_CONTROLLER, OFP_FLOW_PERMANENT, CTRL_PRIORITY), NULL); // Regla para
    procesar los paquetes ARP de otros switches.
    rconn_send(local_rconn, make_add_simple_flow(&tcp_flow, buffer_id,
    OFPP_CONTROLLER, OFP_FLOW_PERMANENT, CTRL_PRIORITY), NULL); // Regla para
    procesar los paquetes TCP de otros switches.

    //Se configuran los drop para evitar bucles con el propio trafico
    local_port_no = get_pw_local_port_number_UAH(in_band->pw); // Se obtiene el
    identificador del puerto fisico compartido con el puerto local.
    netdev_get_in4(in_band->of_device, &local_ip);

    //Se instalan dos reglas para no procesar en la interfaz del puerto local los
    paquetes con MAC origen/destino la de la interfaz del puerto local
    //MAC ORIGEN LA DE LA INTERFAZ DEL PUERTO LOCAL
    local_mac_flow.in_port = htonl(local_port_no);
    memcpy(local_mac_flow.dl_src, netdev_get_etheraddr(in_band->of_device),
    ETH_ADDR_LEN);
    rconn_send(local_rconn, make_add_simple_flow(&local_mac_flow, buffer_id, 0,
    OFP_FLOW_PERMANENT, DROP_PRIORITY), NULL);

    //MAC DESTINO LA DE LA INTERFAZ DEL PUERTO LOCAL
    memset(local_mac_flow.dl_src, 0, ETH_ADDR_LEN);
    memcpy(local_mac_flow.dl_dst, netdev_get_etheraddr(in_band->of_device),
    ETH_ADDR_LEN);
    rconn_send(local_rconn, make_add_simple_flow(&local_mac_flow, buffer_id, 0,
    OFP_FLOW_PERMANENT, DROP_PRIORITY), NULL);
}
```

Código B.17: Obtención flujo del paquete contenido en un PACKET_IN en la función in_band_local_packet_cb() de in_band.c

```
static bool
in_band_local_packet_cb(struct relay *r, void *in_band_)
{
    struct in_band_data *in_band = in_band_;
    struct rconn *rc = r->halves[HALF_LOCAL].rconn;

    struct ofl_msg_packet_in *oflpi;
    struct eth_header *eth;
```

```
    struct ofpbuf *payload, buf_arp, buf_amaru;
    struct ofpbuf *buf;
    uint32_t ip4_aux_1, ip4_aux_2;
    char ip4_1[INET_ADDRSTRLEN], ip4_2[INET_ADDRSTRLEN];
    struct flow flow, flow_inv = {0};
    uint32_t in_port, out_port;

    if (!get_ofp_packet_eth_header_UAH(r, &oflpi, &eth, &in_port, &buf) || !in_band
    ->of_device)
    {
        return false;
    }

    if (oflpi != NULL)
    {
        payload = ofpbuf_new(oflpi->data_length);
        memcpy(payload->data, oflpi->data, oflpi->data_length);

        payload->size = oflpi->data_length;
    }
    else
    {
        return false;
    }
    flow_extract(payload, in_port, &flow);

    ...
```

Código B.18: Procesado del tráfico ARP en la función _in_band_local_packet_cb() de in_band.c

```
if (eth->eth_type == htons(ETH_TYPE_ARP))
    {
        uint32_t buffer_id = 0xffffffff; //NO_BUFFER, para no especificar ningÃºn
    flujo almacenado por el switch
        // Se obtien la cabecera ARP
        struct arp_eth_header *arp;
        buf_arp = *payload;
        eth = ofpbuf_try_pull(&buf_arp, ETH_HEADER_LEN);
        arp = ofpbuf_try_pull(&buf_arp, ARP_ETH_HEADER_LEN);

        if (arp->ar_tpa == rconn_get_ip(in_band->controller) && !eth_addr_equals(
    arp->ar_sha, netdev_get_etheraddr(in_band->of_device))) //Se comprueba si la IP
     buscada es la del Controlador
        {

            in_band_learn_mac(in_band, in_port, eth->eth_src); //Aprende la mac del
     salto anterior

            out_port = get_pw_local_port_number_UAH(in_band->pw); // Se obtiene el
    identificador del puerto fisico que comparte interfaz con el puerto local
```

```
            //Se configura la regla inversa
            flow_inv.dl_type = flow.dl_type;
            memcpy(flow_inv.dl_dst, eth->eth_src, ETH_ADDR_LEN); // MAC switch
    origen como destino
            queue_tx(rc, in_band, make_add_simple_flow(&flow_inv, ntohl(buffer_id),
     in_port, IDLE_ARP_RULE_TIMEOUT, RULE_PRIORITY)); // Regla para el trafico de
    vuelta

            /* If the switch didn't buffer the packet, we need to send a copy. */
            if (ntohl(oflpi->buffer_id) == UINT32_MAX)
            {
                queue_tx(rc, in_band, make_unbuffered_packet_out(payload, in_port,
    out_port));
            }
            else
            {
                queue_tx(rc, in_band, make_buffered_packet_out(oflpi->buffer_id,
    in_port, out_port));
            }
        }
        else
        {
            return false;
        }

...
```

Código B.19: Procesado del tráfico TCP en la función _in_band_local_packet_cb() de in_band.c

```
else if (eth->eth_type == htons(ETH_TYPE_IP) && flow.nw_dst == rconn_get_ip(in_band
    ->controller) && flow.nw_src != local_ip.s_addr)
    {
        out_port = get_pw_local_port_number_UAH(in_band->pw); // Se obtiene el
    identificador del puerto fÃsico que comparte interfaz con el puerto local

        //Se configura la regla en sentido switch --> controlador
        flow_tcp.dl_type = flow.dl_type;
        flow_tcp.nw_proto = flow.nw_proto;
        flow_tcp.nw_src = flow.nw_src;
        flow_tcp.nw_dst = flow.nw_dst;
        flow_tcp.in_port = flow.in_port;


        queue_tx(rc, in_band, make_add_simple_flow(&flow_tcp, ntohl(buffer_id),
    out_port, IDLE_TCP_RULE_TIMEOUT, RULE_PRIORITY)); // Regla para el trafico de
    ida

        memset(&flow_tcp, 0, sizeof(struct flow));
        //Se configura la regla inversa
        flow_tcp.dl_type = flow.dl_type;
        flow_tcp.nw_proto = flow.nw_proto;
```

```
        flow_tcp.nw_dst = flow.nw_src;
        flow_tcp.nw_src = flow.nw_dst;
        queue_tx(rc, in_band, make_add_simple_flow(&flow_tcp, ntohl(buffer_id),
    in_port, IDLE_TCP_RULE_TIMEOUT, RULE_PRIORITY)); // Regla para el trafico de
    vuelta

        /* If the switch didn't buffer the packet, we need to send a copy. */
        if (ntohl(oflpi->buffer_id) == UINT32_MAX)
        {
            queue_tx(rc, in_band, make_unbuffered_packet_out(payload, in_port,
    out_port));
        }
        else
        {
            queue_tx(rc, in_band, make_buffered_packet_out(oflpi->buffer_id,
    in_port, out_port));
        }
    }
    else
    {
        return false;
    }

    ...
```

B.3 Implementación de Amaru

Código B.20: Modificación de la estrucutra reg_AMAC en dp_ports.h

```
struct reg_AMAC
{
    uint8_t level;
    uint8_t AMAC[AMAC_LEN];
    uint16_t port_in;
    uint64_t time_entry;
    bool active; /*Modificacion Boby UAH*/
    struct reg_AMAC *next;
};
```

Código B.21: Modificación de la función table_AMACS_add_AMAC() en dp_ports.c

```
int table_AMACS_add_AMAC(struct table_AMACS *table_AMACS, uint8_t AMAC[AMAC_LEN],
    uint8_t level, uint32_t in_port, int time)
{

    /*Modificaciones Boby UAH*/
    struct reg_AMAC *nuevo_elemento = NULL;
    if ((nuevo_elemento = xmalloc(sizeof(struct reg_AMAC))) == NULL)
    {
        return -1;
```

```
    }
    nuevo_elemento->port_in = in_port;
    nuevo_elemento->time_entry = time_msec() + (time * 1000);
    memcpy(nuevo_elemento->AMAC, AMAC, AMAC_LEN);
    nuevo_elemento->level = level;
    nuevo_elemento->next = NULL;
    nuevo_elemento->active = true;
    if (table_AMACS->num_element == 0)
    {
        table_AMACS->inicio = nuevo_elemento;
        table_AMACS->fin = nuevo_elemento;
    }
    else
    {
        table_AMACS->fin->next = nuevo_elemento; /*Colocamos la nueva AMAC al final
     de la tabla*/
        table_AMACS->fin = nuevo_elemento;
    }
    table_AMACS->num_element++;

    return 0;
    /*+++FIN+++*/
}
```

Código B.22: Modificación de la función dp_ports_output_amaru() en dp_ports.c

```
int dp_ports_output_amaru(struct datapath *dp, struct ofpbuf *buffer UNUSED,
    uint32_t in_port, bool random UNUSED, struct packet *pkt)
{

    struct packet *packet_clone = NULL;
    struct sw_port *p

    LIST_FOR_EACH(p, struct sw_port, node, &dp->port_list)
    {
        if (dp->local_port != NULL)
        {
            if (dp->id == 1 && !strcmp(p->conf->name, dp->local_port->conf->name))
    // Para que el nodo root no envÃe un paquete AMARU al controlador
            {
                continue;
            }
        }
        if (p->conf->port_no == OFPP_LOCAL || p->conf->port_no == in_port)
        {
            continue;
        }

        packet_clone = packet_Amaru(dp, (uint32_t)in_port, false, (pkt->handle_std
    ->proto->amaru->level + 1), p->conf->port_no, pkt->handle_std->proto->amaru->
    amac);
```

```
        VLOG_INFO(LOG_MODULE, "Paquete creado: %s\n", packet_to_string(packet_clone
    ));
        dp_ports_output(dp, packet_clone->buffer, p->conf->port_no, 0); //salgo por
     todos los puertos sin distincion
        log_uah_num_pkt();
        packet_destroy(packet_clone); //limpiamos la memoria reservada para el
    nuevo paquete
    }
        return 0;

}
```

Código B.23: Funciones disable_invalid_amacs_UAH() y enable_invalid_amacs_UAH() en dp_ports.c

```
int disable_invalid_amacs_UAH(struct table_AMACS *table_AMACS, uint32_t down_port)
{
    struct reg_AMAC *aux_AMAC = table_AMACS->inicio;
    while (aux_AMAC != NULL)
    {
        if (aux_AMAC->port_in == down_port && aux_AMAC->active == true)
        {
            aux_AMAC->active = false;
        }

        aux_AMAC = aux_AMAC->next;
    }
    return 0;
}

int enable_valid_amacs_UAH(struct table_AMACS *table_AMACS, uint32_t up_port)
{
    struct reg_AMAC *aux_AMAC = table_AMACS->inicio;
    while (aux_AMAC != NULL)
    {
        if (aux_AMAC->port_in == up_port && aux_AMAC->active == false)
        {
            aux_AMAC->active = true;
        }

        aux_AMAC = aux_AMAC->next;
    }
    return 0;
}
```

Código B.24: Función remove_local_port_UAH() en dp_ports.c

```
struct in_addr remove_local_port_UAH(struct datapath *dp)
{
    int error;
    struct in_addr ip_0 = {INADDR_ANY}, ip_if;                                  //
    Para poner a 0 la ip de la interfaz a eliminar
    netdev_get_in4(dp->local_port->netdev, &ip_if);                             //Se
```

```
    obtiene la ip de la interfaz
    netdev_set_in4(dp->local_port->netdev, ip_0, ip_0);                    //Se
    configura la ip a 0
    error = netdev_set_etheraddr(dp->local_port->netdev, old_local_port_MAC); //Se
    asigna la antigua MAC
    if (error)
    {
        VLOG_WARN(LOG_MODULE, "failed to change %s Ethernet address "
                              "to " ETH_ADDR_FMT ": %s",
                  dp->local_port->conf->name, ETH_ADDR_ARGS(old_local_port_MAC),
    strerror(error));
    }
    list_pop_back(&dp->port_list); //Se elimina el Ãºltimo puerto (puerto_local) de
     la lista

    free(dp->local_port->conf);
    free(dp->local_port->stats);
    free(dp->local_port); //Se libera la memoria del peurto local
    dp->ports_num--;      //Se decrementa el nÃºmero de puertos
    dp->local_port = NULL;

    return ip_if;
}
```

Código B.25: Función configure_new_local_port_amaru_UAH() en dp_ports.c

```
void send_amaru_new_localport_packet_UAH(struct datapath *dp, uint32_t
    new_local_port, char *port_name, struct in_addr *ip, uint32_t *old_local_port)
{
    struct ofl_msg_packet_in msg;
    struct packet *pkt;

    pkt = create_amaru_new_localport_packet_UAH(dp, new_local_port, port_name, ip,
    old_local_port);

    msg.header.type = OFPT_PACKET_IN;
    msg.total_len = pkt->buffer->size;
    msg.reason = OFPR_ACTION;
    msg.table_id = pkt->table_id;
    msg.data = pkt->buffer->data;
    msg.cookie = 0xffffffffffffffff;

    msg.buffer_id = OFP_NO_BUFFER;
    msg.data_length = pkt->buffer->size;

    if (!pkt->handle_std->valid)
    {
        packet_handle_std_validate(pkt->handle_std);
    }
    msg.match = (struct ofl_match_header *)&pkt->handle_std->match;
    dp_send_message(pkt->dp, (struct ofl_msg_header *)&msg, NULL);
```

```
}
```

Código B.26: Función send_amaru_new_localport_packet_UAH() en packet.c

```
int configure_new_local_port_amaru_UAH(struct datapath *dp, struct table_AMACS *
    table_AMACS, struct in_addr *ip, uint32_t old_local_port)
{
    int error;
    struct sw_port *p;
    struct reg_AMAC *aux = table_AMACS->inicio;
    char ip_aux[INET_ADDRSTRLEN];
    struct in_addr mask, local_ip = *ip;
    while (aux != NULL)
    {

        if (aux->active)
        {
            p = dp_ports_lookup(dp, aux->port_in);
            if (netdev_link_state(p->netdev) != NETDEV_LINK_DOWN) //Para comprobar
    si se ha caÃdo el enlace de respaldo durante el proceso de reconfiguracion
            {
            error = dp_ports_add_local(dp, p->conf->name);
            inet_pton(AF_INET, "255.255.255.0", &(mask.s_addr));
            inet_ntop(AF_INET, &local_ip.s_addr, ip_aux, INET_ADDRSTRLEN);
            VLOG_WARN(LOG_MODULE, "[CONFIGURE NEW LOCAL PORT]: Antiguo puerto local
     (%u)\tNuevo puerto local %s(%u)", old_local_port, p->conf->name, p->conf->
    port_no);
            VLOG_WARN(LOG_MODULE, "[CONFIGURE NEW LOCAL PORT]: IP de la interfaz %s
     (mask: %s)", ip_aux, "255.255.255.0");
            netdev_set_in4(dp->local_port->netdev, local_ip, mask); //Se configura
    la ip del nuevo puerto local
            if (error || !netdev_get_in4(dp->local_port->netdev, &local_ip))
            {
                VLOG_WARN(LOG_MODULE, "[CONFIGURE NEW LOCAL PORT]: No se ha
    configurado correctamente el nuevo puero local!!");
                inet_ntop(AF_INET, &local_ip.s_addr, ip_aux, INET_ADDRSTRLEN);
                VLOG_WARN(LOG_MODULE, "[CONFIGURE NEW LOCAL PORT]: IP de la
    interfaz %s", ip_aux);

                return 1;
            }
            send_amaru_new_localport_packet_UAH(dp, p->conf->port_no, p->conf->name
    , ip, &old_local_port); //Se envÃa al ofprotocol el nuevo puerto local
            return 0;
            }
        }
        //actual pasa a ser el siguiente
        aux = aux->next;
    }
    return 1;
}
```

Código B.27: Función create_amaru_new_localport_packet_UAH() en packet.c

```
struct packet *create_amaru_new_localport_packet_UAH(struct datapath *dp, uint32_t
    new_local_port, char *port_name, struct in_addr *ip, uint32_t *old_local_port)
{
    struct packet *pkt = NULL;
    struct ofpbuf *buf = NULL;
    uint8_t char_size;
    struct in_addr local_ip = *ip;
    uint8_t MAC_BC[ETH_ADDR_LEN] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}, type_array
    [2] = {0xAA, 0xAA}; /*Amaru type*/
    FILE *f_packet = fopen("/home/boby/logs/packet.log", "a+");

    //Creamos el buffer del paquete
    buf = ofpbuf_new(LEN_AMARU_PORT_PKT);
    ofpbuf_put(buf, MAC_BC, ETH_ADDR_LEN);
    ofpbuf_put(buf, MAC_BC, ETH_ADDR_LEN);
    ofpbuf_put(buf, type_array, 2);

    //El paquete incorpora el numero del nuevo puerto, el tamano del nombre del
    puerto y el nombre.
    //introducimos el nivel
    ofpbuf_put(buf, &new_local_port, sizeof(uint32_t));
    //Introducimos el tamano del nombre de la interfaz
    char_size = strlen(port_name);
    ofpbuf_put(buf, &char_size, sizeof(char_size));
    //introducimos el nombre de la interfaz
    ofpbuf_put(buf, port_name, strlen(port_name));
    /*Introducimos la ip del puerto local*/
    ofpbuf_put(buf, &local_ip.s_addr, INET_ADDRSTRLEN);
    /*Introducimos el numero del antiguo puerto local*/
    ofpbuf_put(buf, old_local_port, sizeof(uint32_t));
    // //rellenamos para no tener problems con el paquete

    //Creamos el buffer del paquete
    pkt = packet_create(dp, new_local_port, buf, false);

    return pkt;
}
```

Código B.28: Monitorización del estado de los puerto dp_ports_run() en dp_ports.c

```
...

LIST_FOR_EACH_SAFE(p, pn, struct sw_port, node, &dp->port_list)
    {
        int error;
        /* Check for interface state change */
        enum netdev_link_state link_state = netdev_link_state(p->netdev);

        if (link_state == NETDEV_LINK_UP)
        {
```

```
            p->conf->state &= ~OFPPS_LINK_DOWN;
            dp_port_live_update(p);
            /*Modificaciones Boby UAH*/
            if (p->conf->port_no != OFPP_LOCAL)
            {
                enable_valid_amacs_UAH(&table_AMAC, p->conf->port_no); //Se
    reactivan las amacs validas si estaban desactivadas
                visualizar_tabla_AMAC(&table_AMAC, dp->id);
            }
            /*+++FIN+++*/
        }
        else if (link_state == NETDEV_LINK_DOWN)
        {

            p->conf->state |= OFPPS_LINK_DOWN;
            dp_port_live_update(p);

            /*Modificaciones Boby UAH*/
            if (p->conf->port_no == OFPP_LOCAL || ((time_msec() -
    time_init_local_port) < 5000)) // Se comprueba si se trata del puerto local, si
     no se ha cumplido el tiempo de espera para la reconfiguracion y si no esta
    configurado el       puerto local.
            {
                continue;
            }


            disable_invalid_amacs_UAH(&table_AMAC, p->conf->port_no); //Se
    desactivan las AMACs asociadas al puerto que se ha caido.
            visualizar_tabla_AMAC(&table_AMAC, dp->id);

            if (dp->local_port != NULL && local_port_ok)
            {
                if (!strcmp(p->conf->name, dp->local_port->conf->name) && (dp->id
    != 1))
                {
                    struct in_addr ip_if;
                    uint32_t old_local_port;

                    local_port_ok = false;
                    old_local_port = p->conf->port_no;
                    ip_if = remove_local_port_UAH(dp);
                    configure_new_local_port_amaru_UAH(dp, &table_AMAC, &ip_if,
    old_local_port);
                    VLOG_WARN(LOG_MODULE, "[DP PORTS RUN]: Se ha configurado el
    nuevo puerto local >> %s<<", dp->local_port->conf->name);
                    time_init_local_port = 0;
                }
            }
            /*+++FIN+++*/
...
```

Código B.29: Validación de la configuración del nuevo puerto local en dp_ports_run() en dp_ports.c

```
...

error = netdev_recv(p->netdev, buffer, VLAN_ETH_HEADER_LEN + max_mtu);
        if (!error)
        {
            p->stats->rx_packets++;
            p->stats->rx_bytes += buffer->size;
            process_buffer(dp, p, buffer);
            buffer = NULL;

            /*Modificaciones Boby UAH*/
            /*Se comprueba si se ha recibido paquetes en la interfaz configurada
    como puerto local para poder dar por finalizada la configuracion del puerto
    local*/
            if (dp->local_port != NULL && !strcmp(p->conf->name, dp->local_port->
    conf->name))
            {

                link_state = netdev_link_state(dp->local_port->netdev);
                if (link_state != NETDEV_LINK_DOWN && !local_port_ok)
                {

                    local_port_ok = true; //Si se ha recibido paquetes a traves de
    la interfac configurada como nuevo puerto local
                                          //se considera que ha finalizado la
    cofniguracion del nuevo puerto local
                }
            }
            /*++++FIN++++/

...
```

Código B.30: Función install_new_localport_rules_UAH() en in_band.c

```
void install_new_localport_rules_UAH(struct rconn *local_rconn, uint32_t *
    new_local_port, struct in_addr *local_ip, struct in_addr *controller_ip,
    uint32_t *old_local_port)
{
    uint32_t buffer_id = 0xffffffff;
    struct flow tcp_flow = {0}, tcp_mod_flow = {0}, tcp_del_flow = {0};
    char ip_char[INET_ADDRSTRLEN];

    // Drop TCP con origen/destino el Switch teniendo en cuenta el nuevo puerto
    inet_ntop(AF_INET, &local_ip->s_addr, ip_char, INET_ADDRSTRLEN);
    VLOG_WARN(LOG_MODULE, "[INSTALL NEW LOCALPORT RULES]: IP: %s\t Puerto: %u",
    ip_char, *new_local_port);
    tcp_flow.dl_type = htons(ETH_TYPE_IP);
    tcp_flow.nw_proto = IP_TYPE_TCP;
```

```
    tcp_flow.nw_dst = local_ip->s_addr;
    tcp_flow.in_port = htonl(*new_local_port);
    rconn_send(local_rconn, make_add_simple_flow(&tcp_flow, buffer_id, 0,
    OFP_FLOW_PERMANENT, DROP_PRIORITY), NULL);

    tcp_flow.nw_src = local_ip->s_addr;
    tcp_flow.nw_dst = 0;
    tcp_flow.in_port = htonl(*new_local_port);
    rconn_send(local_rconn, make_add_simple_flow(&tcp_flow, buffer_id, 0,
    OFP_FLOW_PERMANENT, DROP_PRIORITY), NULL);

    /*Se eliminan los flujos TCP con destino el controlador del antiguo puerto
    local*/
    tcp_mod_flow.dl_type = htons(ETH_TYPE_IP);
    tcp_mod_flow.nw_proto = IP_TYPE_TCP;
    tcp_mod_flow.nw_dst = controller_ip->s_addr;
    rconn_send(local_rconn, make_del_flow(&tcp_mod_flow, 0x00), NULL);

    // Se eliminan los flujos DROP para el antiguo puerto local
    tcp_del_flow.dl_type = htons(ETH_TYPE_IP);
    tcp_del_flow.nw_proto = IP_TYPE_TCP;
    tcp_del_flow.nw_dst = local_ip->s_addr;
    tcp_del_flow.in_port = htonl(*old_local_port);
    rconn_send(local_rconn, make_del_flow(&tcp_del_flow, 0x00), NULL);

    tcp_del_flow.nw_src = local_ip->s_addr;
    tcp_del_flow.nw_dst = 0;
    rconn_send(local_rconn, make_del_flow(&tcp_del_flow, 0x00), NULL);
}
```

Código B.31: Procesado del paquete Amaru en la función _in_band_local_packet_cb() de in_band.c

```
...
//Manejamos el paquete amaru que indica el neuvo puerto local
    if (eth->eth_type == htons(ETH_TYPE_AMARU))
    {
        uint32_t *new_local_port, *old_local_port;
        uint8_t *char_size;
        char *port_name, ip_char[INET_ADDRSTRLEN];
        struct in_addr *local_ip_amaru, controller_ip;
        buf_amaru = *payload;
        ofpbuf_try_pull(&buf_amaru, ETH_HEADER_LEN);                    //Nos
    deshacemos de la cabecera ethernet
        new_local_port = ofpbuf_try_pull(&buf_amaru, sizeof(uint32_t)); //Se
    obtiene el numero del nuevo puerto local
        char_size = ofpbuf_try_pull(&buf_amaru, sizeof(uint8_t));       //Se
    obtiene el tamano del nombre del puerto
        port_name = ofpbuf_try_pull(&buf_amaru, *char_size);            //Se
    obtiene el nombre del nuevo puerto local
        local_ip_amaru = ofpbuf_try_pull(&buf_amaru, INET_ADDRSTRLEN);  //Se
    obtiene la ip del puerto local
```

```
        old_local_port = ofpbuf_try_pull(&buf_amaru, sizeof(uint32_t)); //Se
    obtiene el numero del antiguo puerto local

        inet_ntop(AF_INET, &local_ip_amaru->s_addr, ip_char, INET_ADDRSTRLEN);
        controller_ip.s_addr = rconn_get_ip(in_band->controller);
        install_new_localport_rules_UAH(r->halves[HALF_LOCAL].rconn, new_local_port
    , local_ip_amaru, &controller_ip, old_local_port);
        return false; // Para que no envie el packet in al controlador.
    }

...
```

Anexo C

Código generado para los escenarios de pruebas

Código C.1: Script bash del escenario 1

```
#!/bin/sh
##Se crean los espacios de nombres para los host
ip netns add host1
ip netns add host2

##Se crean las interfaces veth
ip link add veth-s12 type veth peer name veth-s21
ip link add veth-s23 type veth peer name veth-s32
ip link add eth0-h1 type veth peer name veth-h1
ip link add eth0-h2 type veth peer name veth-h2

##Se asignan los interfaces a los namespaces
ip link set eth0-h1 netns host1
ip link set eth0-h2 netns host2


##Se configuran los interfaces del switch1
ip link set veth-s12 up

##Se configuran los interfaces del switch2
ip link set veth-s21 up
ip link set veth-s23 up
ip link set veth-h1 up

##Se configuran los interfaces del switch3
ip link set veth-s32 up
ip link set veth-h2 up

##Se configuran los interfaces de los host
ip netns exec host1 ifconfig eth0-h1 10.0.0.11/24 up
ip netns exec host1 ethtool -K eth0-h1 tx off #Para desactivar el offloading de
    tx y no tener problemas con los checksum de TCP!!!
```

```
ip netns exec host2 ifconfig eth0-h2 10.0.0.12/24 up
ip netns exec host2 ethtool -K eth0-h2 tx off
```

Código C.2: Script bash del escenario 2

```
#!/bin/sh
#Se crean los espacios de nombres
ip netns add switch1
ip netns add switch2
ip netns add host1
ip netns add host2

##Se crean los interfaces veth
ip link add veth-ctrl type veth peer name veth-s1
ip link add veth-s12 type veth peer name veth-s21
ip link add veth-s23 type veth peer name veth-s32
ip link add eth0-h1 type veth peer name veth-h1
ip link add eth0-h2 type veth peer name veth-h2

##Se asignan los interfaces a los namespaces
ip link set veth-s1 netns switch1
ip link set veth-s12 netns switch1


ip link set veth-s23 netns switch2
ip link set veth-s21 netns switch2
ip link set veth-h1 netns switch2
ip link set eth0-h1 netns host1

ip link set eth0-h2 netns host2


##Se configura el interfaz del controlador
ifconfig veth-ctrl 10.0.0.88/24 up
ethtool -K veth-ctrl tx off #Para desactivar el offloading de tx y no tener
    problemas con los checksum de TCP!!!

##Se configuran los interfaces del switch1
ip netns exec switch1 ifconfig veth-s1 10.0.0.101/24 up
ip netns exec switch1 ip link set veth-s12 up
ip netns exec switch1 ethtool -K veth-s1 tx off

##Se configuran los interfaces del switch2
ip netns exec switch2 ifconfig veth-s21 10.0.0.102/24 up
ip netns exec switch2 ip link set veth-s23 up
ip netns exec switch2 ip link set veth-h1 up
ip netns exec switch2 ethtool -K veth-s21 tx off
ip netns exec switch2 ethtool -K veth-s23 tx off

##Se configuran los interfaces del switch3
ip link set veth-s32 up
ip link set veth-h2 up
```

```
47
48 ##Se configuran los interfaces de los host
49 ip netns exec host1 ifconfig eth0-h1 10.0.0.11/24 up
50 ip netns exec host1 ethtool -K eth0-h1 tx off
51 ip netns exec host2 ifconfig eth0-h2 10.0.0.12/24 up
52 ip netns exec host2 ethtool -K eth0-h2 tx off
```

Código C.3: Script bash del escenario 3 con nivel de conexión 2

```
1  #!/bin/sh
2  #Se crean los espacios de nombres
3  ip netns add switch1
4  ip netns add switch2
5  ip netns add switch3
6  ip netns add switch4
7  ip netns add host1
8  ip netns add host2
9  ip netns add host3
10
11 ##Se crean los interfaces veth
12 ip link add veth-ctrl type veth peer name veth-s1
13 ip link add veth-s12 type veth peer name veth-s21
14 ip link add veth-s13 type veth peer name veth-s31
15 ip link add veth-s24 type veth peer name veth-s42
16 ip link add veth-s34 type veth peer name veth-s43
17 ip link add eth0-h1 type veth peer name veth-h1
18 ip link add eth0-h2 type veth peer name veth-h2
19 ip link add eth0-h3 type veth peer name veth-h3
20
21 ##Se asignan los interfaces a los namespaces
22 ip link set veth-s1 netns switch1
23 ip link set veth-s12 netns switch1
24 ip link set veth-s13 netns switch1
25
26 ip link set veth-s21 netns switch2
27 ip link set veth-s24 netns switch2
28 ip link set veth-h1 netns switch2
29 ip link set eth0-h1 netns host1
30
31 ip link set veth-s31 netns switch3
32 ip link set veth-s34 netns switch3
33 ip link set veth-h2 netns switch3
34 ip link set eth0-h2 netns host2
35
36 ip link set veth-s42 netns switch4
37 ip link set veth-s43 netns switch4
38 ip link set veth-h3 netns switch4
39 ip link set eth0-h3 netns host3
40
41 ##Se configura el interfaz del controlador
42 ifconfig veth-ctrl 10.0.0.88/24 up
43 ethtool -K veth-ctrl tx off #Para desactivar el offloading de tx y no tener
```

```
    problemas con los checksum de TCP!!!

##Se configuran los interfaces del switch1
ip netns exec switch1 ifconfig veth-s1 10.0.0.101/24 up
ip netns exec switch1 ip link set veth-s12 up
ip netns exec switch1 ip link set veth-s13 up

##Se configuran los interfaces del switch2
ip netns exec switch2 ifconfig veth-s21 10.0.0.102/24 up
ip netns exec switch2 ip link set veth-h1 up
ip netns exec switch2 ip link set veth-s24 up
ip netns exec switch2 ethtool -K veth-s21 tx off
ip netns exec switch2 ethtool -K veth-s24 tx off

##Se configuran los interfaces del switch3
ip netns exec switch3 ifconfig veth-s31 10.0.0.103/24 up
ip netns exec switch3 ip link set veth-h2 up
ip netns exec switch3 ip link set veth-s34 up
ip netns exec switch3 ethtool -K veth-s31 tx off
ip netns exec switch3 ethtool -K veth-s34 tx off

##Se configuran los interfaces del switch4
ip netns exec switch4 ifconfig veth-s42 10.0.0.104/24 up
ip netns exec switch4 ip link set veth-h3 up
ip netns exec switch4 ip link set veth-s43 up
ip netns exec switch4 ethtool -K veth-s42 tx off
ip netns exec switch4 ethtool -K veth-s43 tx off

##Se configuran los interfaces de los host
ip netns exec host1 ifconfig eth0-h1 10.0.0.11/24 up
ip netns exec host1 ethtool -K eth0-h1 tx off
ip netns exec host2 ifconfig eth0-h2 10.0.0.12/24 up
ip netns exec host2 ethtool -K eth0-h2 tx off
ip netns exec host3 ifconfig eth0-h3 10.0.0.13/24 up
ip netns exec host3 ethtool -K eth0-h3 tx off
```

Código C.4: Script bash del escenario 3 con nivel de conexión 3

```
#!/bin/sh
#Se crean los espacios de nombres
ip netns add switch1
ip netns add switch2
ip netns add switch3
ip netns add switch4
ip netns add host1
ip netns add host2
ip netns add host3

##Se crean los interfaces veth
ip link add veth-ctrl type veth peer name veth-s1
ip link add veth-s12 type veth peer name veth-s21
ip link add veth-s13 type veth peer name veth-s31
```

```
ip link add veth-s23 type veth peer name veth-s32
ip link add veth-s24 type veth peer name veth-s42
ip link add veth-s34 type veth peer name veth-s43
ip link add veth-s14 type veth peer name veth-s41
ip link add eth0-h1 type veth peer name veth-h1
ip link add eth0-h2 type veth peer name veth-h2
ip link add eth0-h3 type veth peer name veth-h3

##Se asignan los interfaces a los namespaces
ip link set veth-s1 netns switch1
ip link set veth-s12 netns switch1
ip link set veth-s13 netns switch1
ip link set veth-s14 netns switch1

ip link set veth-s21 netns switch2
ip link set veth-s23 netns switch2
ip link set veth-s24 netns switch2
ip link set veth-h1 netns switch2
ip link set eth0-h1 netns host1

ip link set veth-s31 netns switch3
ip link set veth-s32 netns switch3
ip link set veth-s34 netns switch3
ip link set veth-h2 netns switch3
ip link set eth0-h2 netns host2

ip link set veth-s41 netns switch4
ip link set veth-s42 netns switch4
ip link set veth-s43 netns switch4
ip link set veth-h3 netns switch4
ip link set eth0-h3 netns host3

##Se configura el interfaz del controlador
ifconfig veth-ctrl 10.0.0.88/24 up
ethtool -K veth-ctrl tx off #Para desactivar el offloading de tx y no tener
    problemas con los checksum de TCP!!!
##Se configuran los interfaces del switch1
ip netns exec switch1 ifconfig veth-s1 10.0.0.101/24 up
ip netns exec switch1 ip link set veth-s12 up
ip netns exec switch1 ip link set veth-s13 up
ip netns exec switch1 ip link set veth-s14 up

##Se configuran los interfaces del switch2
ip netns exec switch2 ifconfig veth-s21 10.0.0.102/24 up
ip netns exec switch2 ip link set veth-h1 up
ip netns exec switch2 ip link set veth-s23 up
ip netns exec switch2 ip link set veth-s24 up
ip netns exec switch2 ethtool -K veth-s21 tx off
ip netns exec switch2 ethtool -K veth-s23 tx off
ip netns exec switch2 ethtool -K veth-s24 tx off

```

```
65 ##Se configuran los interfaces del switch3
66 ip netns exec switch3 ifconfig veth-s31 10.0.0.103/24 up
67 ip netns exec switch3 ip link set veth-h2 up
68 ip netns exec switch3 ip link set veth-s32 up
69 ip netns exec switch3 ip link set veth-s34 up
70 ip netns exec switch3 ethtool -K veth-s31 tx off
71 ip netns exec switch3 ethtool -K veth-s32 tx off
72 ip netns exec switch3 ethtool -K veth-s34 tx off
73
74 ##Se configuran los interfaces del switch4
75 ip netns exec switch4 ifconfig veth-s42 10.0.0.104/24 up
76 ip netns exec switch4 ip link set veth-h3 up
77 ip netns exec switch4 ip link set veth-s41 up
78 ip netns exec switch4 ip link set veth-s43 up
79 ip netns exec switch4 ethtool -K veth-s41 tx off
80 ip netns exec switch4 ethtool -K veth-s42 tx off
81 ip netns exec switch4 ethtool -K veth-s43 tx off
82
83 ##Se configuran los interfaces de los host
84 ip netns exec host1 ifconfig eth0-h1 10.0.0.11/24 up
85 ip netns exec host1 ethtool -K eth0-h1 tx off
86 ip netns exec host2 ifconfig eth0-h2 10.0.0.12/24 up
87 ip netns exec host2 ethtool -K eth0-h2 tx off
88 ip netns exec host3 ifconfig eth0-h3 10.0.0.13/24 up
89 ip netns exec host3 ethtool -K eth0-h3 tx off
```

Código C.5: Script bash del escenario rombo

```
1  #!/bin/sh
2  #Se crean los espacios de nombres
3  ip netns add switch1
4  ip netns add switch2
5  ip netns add switch3
6  ip netns add switch4
7  ip netns add host1
8  ip netns add host2
9  ip netns add host3
10
11
12 ##Se crean los interfaces veth
13 ip link add veth-ctrl type veth peer name veth-s1
14 ip link add veth-s12 type veth peer name veth-s21
15 ip link add veth-s13 type veth peer name veth-s31
16 ip link add veth-s23 type veth peer name veth-s32
17 ip link add veth-s24 type veth peer name veth-s42
18 ip link add veth-s34 type veth peer name veth-s43
19 ip link add eth0-h1 type veth peer name veth-h1
20 ip link add eth0-h2 type veth peer name veth-h2
21 ip link add eth0-h3 type veth peer name veth-h3
22
23
24 ##Se asignan los interfaces a los namespaces
```

```
25 ip link set veth-s1 netns switch1
26 ip link set veth-s12 netns switch1
27 ip link set veth-s13 netns switch1
28
29 ip link set veth-s21 netns switch2
30 ip link set veth-s23 netns switch2
31 ip link set veth-s24 netns switch2
32 ip link set veth-h1 netns switch2
33 ip link set eth0-h1 netns host1
34
35 ip link set veth-s31 netns switch3
36 ip link set veth-s32 netns switch3
37 ip link set veth-s34 netns switch3
38 ip link set veth-h2 netns switch3
39 ip link set eth0-h2 netns host2
40
41 ip link set veth-s42 netns switch4
42 ip link set veth-s43 netns switch4
43 ip link set veth-h3 netns switch4
44 ip link set eth0-h3 netns host3
45
46 ##Se configura el interfaz del controlador
47 ifconfig veth-ctrl 10.0.0.88/24 up
48 ethtool -K veth-ctrl tx off #Para desactivar el offloading de tx y no tener
       problemas con los checksum de TCP!!!
49
50 ##Se configuran los interfaces del switch1
51
52 ip netns exec switch1 ifconfig veth-s1 10.0.0.101/24 up
53 ip netns exec switch1 ip link set veth-s12 up
54 ip netns exec switch1 ip link set veth-s13 up
55
56
57 ##Se configuran los interfaces del switch2
58 ip netns exec switch2 ifconfig veth-s21 10.0.0.102/24 up
59 ip netns exec switch2 ip link set veth-h1 up
60 ip netns exec switch2 ip link set veth-s23 up
61 ip netns exec switch2 ip link set veth-s24 up
62 ip netns exec switch2 ethtool -K veth-s21 tx off
63 ip netns exec switch2 ethtool -K veth-s23 tx off
64 ip netns exec switch2 ethtool -K veth-s24 tx off
65
66 ##Se configuran los interfaces del switch3
67 ip netns exec switch3 ifconfig veth-s31 10.0.0.103/24 up
68 ip netns exec switch3 ip link set veth-h2 up
69 ip netns exec switch3 ip link set veth-s32 up
70 ip netns exec switch3 ip link set veth-s34 up
71 ip netns exec switch3 ethtool -K veth-s31 tx off
72 ip netns exec switch3 ethtool -K veth-s32 tx off
73 ip netns exec switch3 ethtool -K veth-s34 tx off
74
```

```
75 ##Se configuran los interfaces del switch4
76 ip netns exec switch4 ifconfig veth-s42 10.0.0.104/24 up
77 ip netns exec switch4 ip link set veth-h3 up
78 ip netns exec switch4 ip link set veth-s43 up
79 ip netns exec switch4 ethtool -K veth-s42 tx off
80 ip netns exec switch4 ethtool -K veth-s43 tx off
81
82 ##Se configuran los interfaces de los host
83 ip netns exec host1 ifconfig eth0-h1 10.0.0.11/24 up
84 ip netns exec host1 ethtool -K eth0-h1 tx off
85 ip netns exec host2 ifconfig eth0-h2 10.0.0.12/24 up
86 ip netns exec host2 ethtool -K eth0-h2 tx off
87 ip netns exec host3 ifconfig eth0-h3 10.0.0.13/24 up
88 ip netns exec host3 ethtool -K eth0-h3 tx off
```

Código C.6: Script bash del escenario triple rombo

```
1  #!/bin/sh
2  #Se crean los espacios de nombres
3  ip netns add switch1
4  ip netns add switch2
5  ip netns add switch3
6  ip netns add switch4
7  ip netns add switch5
8  ip netns add switch6
9
10 ip netns add host1
11 ip netns add host2
12 ip netns add host3
13 ip netns add host4
14 ip netns add host5
15
16
17 ##Se crean los interfaces veth
18 ip link add veth-ctrl type veth peer name veth-s1
19 ip link add veth-s12 type veth peer name veth-s21
20 ip link add veth-s13 type veth peer name veth-s31
21 ip link add veth-s23 type veth peer name veth-s32
22 ip link add veth-s24 type veth peer name veth-s42
23 ip link add veth-s25 type veth peer name veth-s52
24 ip link add veth-s34 type veth peer name veth-s43
25 ip link add veth-s36 type veth peer name veth-s63
26 ip link add veth-s45 type veth peer name veth-s54
27 ip link add veth-s46 type veth peer name veth-s64
28
29 ip link add eth0-h1 type veth peer name veth-h1
30 ip link add eth0-h2 type veth peer name veth-h2
31 ip link add eth0-h3 type veth peer name veth-h3
32 ip link add eth0-h4 type veth peer name veth-h4
33 ip link add eth0-h5 type veth peer name veth-h5
34
35
```

```
##Se asignan los interfaces a los namespaces
ip link set veth-s1 netns switch1
ip link set veth-s12 netns switch1
ip link set veth-s13 netns switch1


ip link set veth-s21 netns switch2
ip link set veth-s23 netns switch2
ip link set veth-s24 netns switch2
ip link set veth-s25 netns switch2
ip link set veth-h1 netns switch2
ip link set eth0-h1 netns host1

ip link set veth-s31 netns switch3
ip link set veth-s32 netns switch3
ip link set veth-s34 netns switch3
ip link set veth-s36 netns switch3
ip link set veth-h2 netns switch3
ip link set eth0-h2 netns host2

ip link set veth-s42 netns switch4
ip link set veth-s43 netns switch4
ip link set veth-s45 netns switch4
ip link set veth-s46 netns switch4
ip link set veth-h3 netns switch4
ip link set eth0-h3 netns host3

ip link set veth-s52 netns switch5
ip link set veth-s54 netns switch5
ip link set veth-h4 netns switch5
ip link set eth0-h4 netns host4

ip link set veth-s63 netns switch6
ip link set veth-s64 netns switch6
ip link set veth-h5 netns switch6
ip link set eth0-h5 netns host5

##Se configura el interfaz del controlador
ifconfig veth-ctrl 10.0.0.88/24 up
ethtool -K veth-ctrl tx off #Para desactivar el offloading de tx y no tener
    problemas con los checksum de TCP!!!

##Se configuran los interfaces del switch1
ip netns exec switch1 ifconfig veth-s1 10.0.0.101/24 up
ip netns exec switch1 ip link set veth-s12 up
ip netns exec switch1 ip link set veth-s13 up

##Se configuran los interfaces del switch2
ip netns exec switch2 ifconfig veth-s21 10.0.0.102/24 up
ip netns exec switch2 ip link set veth-h1 up
ip netns exec switch2 ip link set veth-s23 up
```

```
ip netns exec switch2 ip link set veth-s24 up
ip netns exec switch2 ip link set veth-s25 up
ip netns exec switch2 ethtool -K veth-s21 tx off
ip netns exec switch2 ethtool -K veth-s23 tx off
ip netns exec switch2 ethtool -K veth-s24 tx off
ip netns exec switch2 ethtool -K veth-s25 tx off

##Se configuran los interfaces del switch3
ip netns exec switch3 ifconfig veth-s31 10.0.0.103/24 up
ip netns exec switch3 ip link set veth-h2 up
ip netns exec switch3 ip link set veth-s32 up
ip netns exec switch3 ip link set veth-s34 up
ip netns exec switch3 ip link set veth-s36 up
ip netns exec switch3 ethtool -K veth-s31 tx off
ip netns exec switch3 ethtool -K veth-s32 tx off
ip netns exec switch3 ethtool -K veth-s34 tx off
ip netns exec switch3 ethtool -K veth-s36 tx off

##Se configuran los interfaces del switch4
ip netns exec switch4 ifconfig veth-s42 10.0.0.104/24 up
ip netns exec switch4 ip link set veth-h3 up
ip netns exec switch4 ip link set veth-s43 up
ip netns exec switch4 ip link set veth-s45 up
ip netns exec switch4 ip link set veth-s46 up
ip netns exec switch4 ethtool -K veth-s42 tx off
ip netns exec switch4 ethtool -K veth-s43 tx off
ip netns exec switch4 ethtool -K veth-s45 tx off
ip netns exec switch4 ethtool -K veth-s46 tx off

##Se configuran los interfaces del switch5
ip netns exec switch5 ifconfig veth-s52 10.0.0.105/24 up
ip netns exec switch5 ip link set veth-h4 up
ip netns exec switch5 ip link set veth-s54 up
ip netns exec switch5 ethtool -K veth-s52 tx off
ip netns exec switch5 ethtool -K veth-s54 tx off

##Se configuran los interfaces del switch6
ip netns exec switch6 ifconfig veth-s63 10.0.0.106/24 up
ip netns exec switch6 ip link set veth-h5 up
ip netns exec switch6 ip link set veth-s64 up
ip netns exec switch6 ethtool -K veth-s63 tx off
ip netns exec switch6 ethtool -K veth-s64 tx off

##Se configuran los interfaces de los host
ip netns exec host1 ifconfig eth0-h1 10.0.0.11/24 up
ip netns exec host1 ethtool -K eth0-h1 tx off
ip netns exec host2 ifconfig eth0-h2 10.0.0.12/24 up
ip netns exec host2 ethtool -K eth0-h2 tx off
ip netns exec host3 ifconfig eth0-h3 10.0.0.13/24 up
ip netns exec host3 ethtool -K eth0-h3 tx off
ip netns exec host4 ifconfig eth0-h4 10.0.0.14/24 up
```

```
ip netns exec host4 ethtool -K eth0-h4 tx off
ip netns exec host5 ifconfig eth0-h5 10.0.0.15/24 up
ip netns exec host5 ethtool -K eth0-h5 tx off
```

Código C.7: Script bash para levantar los switches del escenario 1

```
#!/bin/sh

## Switch 1
ofdatapath --detach -i veth-s12 punix:/tmp/s1 -d 000000000001 --no-slicing
ofprotocol --out-of-band unix:/tmp/s1 tcp:127.0.0.1:6653 --fail=closed --listen=
    punix:/tmp/s1.listen

## Switch 2
ofdatapath --detach -i veth-s21,veth-s23,veth-h1 punix:/tmp/s2 -d 000000000002
    --no-slicing
ofprotocol --out-of-band unix:/tmp/s2 tcp:127.0.0.1:6653 --fail=closed --listen=
    punix:/tmp/s2.listen

## Switch 3
ofdatapath --detach -i veth-s32,veth-h2 punix:/tmp/s3 -d 000000000003 --no-
    slicing
ofprotocol --out-of-band unix:/tmp/s3 tcp:127.0.0.1:6653 --fail=closed --listen=
    punix:/tmp/s3.listen
```

Código C.8: Script bash para levantar los switches del escenario 2

```
#!/bin/sh

## Switch 1
ofdatapath --detach -i veth-s1,veth-s12 punix:/tmp/s1 -d 000000000001 --local-
    port=veth-s1 --no-slicing
ofprotocol unix:/tmp/s1 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s1.
    listen

## Switch 2
ofdatapath --detach -i veth-s21,veth-s23,veth-h1 punix:/tmp/s2 -d 000000000002
    --local-port=veth-s21 --no-slicing
ofprotocol unix:/tmp/s2 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s2.
    listen

## Switch 3
ofdatapath --detach -i veth-s32,veth-h2 punix:/tmp/s3 -d 000000000003 --no-
    slicing
ofprotocol --out-of-band unix:/tmp/s3 tcp:127.0.0.1:6653 --fail=closed --listen=
    punix:/tmp/s3.listen
```

Código C.9: Script bash para levantar los switches del escenario 3 con nivel de conexión 2

```
#!/bin/sh

## Switch 1
ofdatapath --detach -i veth-s1,veth-s12,veth-s13 punix:/tmp/s1 -d 000000000001
```

```
    --local-port=veth-s1 --no-slicing
ofprotocol unix:/tmp/s1 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s1.
    listen

## Switch 2
ofdatapath --detach -i veth-s21,veth-s24,veth-h1 punix:/tmp/s2 -d 000000000002
    --local-port=veth-s21 --no-slicing
ofprotocol unix:/tmp/s2 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s2.
    listen

## Switch 3
ofdatapath --detach -i veth-s31,veth-s34,veth-h2 punix:/tmp/s3 -d 000000000003
    --local-port=veth-s31 --no-slicing
ofprotocol unix:/tmp/s3 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s3.
    listen

## Switch 4
ofdatapath --detach -i veth-s42,veth-s43,veth-h3 punix:/tmp/s4 -d 000000000004
    --local-port=veth-s42 --no-slicing
ofprotocol unix:/tmp/s4 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s4.
    listen
```

Código C.10: Script bash para levantar los switches del escenario 3 con nivel de conexión 3

```
#!/bin/sh

## Switch 1
ofdatapath --detach -i veth-s1,veth-s12,veth-s13,veth-s14 punix:/tmp/s1 -d
    000000000001 --local-port=veth-s1 --no-slicing
ofprotocol unix:/tmp/s1 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s1.
    listen

## Switch 2
ofdatapath --detach -i veth-s21,veth-s23,veth-s24,veth-h1 punix:/tmp/s2 -d
    000000000002 --local-port=veth-s21 --no-slicing
ofprotocol unix:/tmp/s2 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s2.
    listen

## Switch 3
ofdatapath --detach -i veth-s31,veth-s32,veth-s34,veth-h2 punix:/tmp/s3 -d
    000000000003 --local-port=veth-s31 --no-slicing
ofprotocol unix:/tmp/s3 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s3.
    listen

## Switch 4
ofdatapath --detach -i veth-s41,veth-s42,veth-s43,veth-h3 punix:/tmp/s4 -d
    000000000004 --local-port=veth-s42 --no-slicing
ofprotocol unix:/tmp/s4 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s4.
    listen
```

Código C.11: Script bash para levantar los switches del escenario rombo

```
#!/bin/sh

## Switch 1
ofdatapath --detach -i veth-s1,veth-s12,veth-s13 punix:/tmp/s1 -d 000000000001
    --local-port=veth-s1 --no-slicing
ofprotocol unix:/tmp/s1 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s1.
    listen

## Switch 2
ofdatapath --detach -i veth-s21,veth-s23,veth-s24,veth-h1 punix:/tmp/s2 -d
    000000000002 --local-port=veth-s21 --no-slicing
ofprotocol unix:/tmp/s2 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s2.
    listen

## Switch 3
ofdatapath --detach -i veth-s31,veth-s32,veth-s34,veth-h2 punix:/tmp/s3 -d
    000000000003 --local-port=veth-s31 --no-slicing
ofprotocol unix:/tmp/s3 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s3.
    listen

## Switch 4
ofdatapath --detach -i veth-s42,veth-s43,veth-h3 punix:/tmp/s4 -d 000000000004
    --local-port=veth-s42 --no-slicing
ofprotocol unix:/tmp/s4 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s4.
    listen
```

Código C.12: Script bash para levantar los switches del escenario triple rombo

```
#!/bin/sh

## Switch 1
ofdatapath --detach -i veth-s1,veth-s12,veth-s13 punix:/tmp/s1 -d 000000000001
    --local-port=veth-s1 --no-slicing
ofprotocol unix:/tmp/s1 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s1.
    listen

## Switch 2
ofdatapath --detach -i veth-s21,veth-s23,veth-s24,veth-s25,veth-h1 punix:/tmp/s2
     -d 000000000002 --local-port=veth-s21 --no-slicing
ofprotocol unix:/tmp/s2 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s2.
    listen

## Switch 3
ofdatapath --detach -i veth-s31,veth-s32,veth-s34,veth-s36,veth-h2 punix:/tmp/s3
     -d 000000000003 --local-port=veth-s31 --no-slicing
ofprotocol unix:/tmp/s3 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s3.
    listen

## Switch 4
ofdatapath --detach -i veth-s42,veth-s43,veth-s45,veth-s46,veth-h3 punix:/tmp/s4
     -d 000000000004 --local-port=veth-s42 --no-slicing
ofprotocol unix:/tmp/s4 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s4.
```

```
    listen

## Switch 5
ofdatapath --detach -i veth-s52,veth-s54,veth-h4 punix:/tmp/s5 -d 000000000005
    --local-port=veth-s52 --no-slicing
ofprotocol unix:/tmp/s5 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s5.
    listen

## Switch 6
ofdatapath --detach -i veth-s63,veth-s64,veth-h5 punix:/tmp/s6 -d 000000000006
    --local-port=veth-s63 --no-slicing
ofprotocol unix:/tmp/s6 tcp:10.0.0.88:6653 --fail=closed --listen=punix:/tmp/s6.
    listen
```

Anexo D

Pliego de condiciones

En este anexo se proporciona los elementos *hardware* y *software* utilizados para el desarrollo del proyecto. Cabe destacar que durante la realización del proyecto, se ha utilizado un único equipo tanto para llevar a cabo el desarrollo de las implementaciones, como para realizar las pruebas. Las principales características del equipo son las siguientes:

- HP ELITEONE G3:
  - Procesador: Intel Core i5-7500 (4 núcleos) 3.4GHz
  - Memoria RAM: 32 GB
  - Memoria ROM: 256 GB
  - GPU: Intel HD Graphics 630 (Kaby Lake GT2)

  El equipo cuenta con el siguiente *software*:

- Sistema Operativo Ubuntu Desktop 16.04 LTS cuya versión del *kernel* es 4.15.0-132-generic.
- Paquete de herramientas `iproute2` para configurar `network namespaces` e interfaces `ethernet` virtuales.
- Visual Studio Code para programar y depurar el switch BOFUSS.
- Herramienta Wireshark para capturar los paquetes que se transmiten a través de las interfaces.
- Código abierto del *software-switch* BOFUSS (OfSoftSwitch 13).

Anexo E

Presupuesto

En este apéndice se desglosan los diferentes costes asociados al proyecto. Se ha dividido en 3 apartados en los que se proporcionan los costes del material, los costes de personal y el presupuesto final, respectivamente.

E.1 Costes del material

Los costes asociados al equipamiento utilizado durante el desarrollo del trabajo se muestran en la Tabla E.1.

Concepto	**Precio**	**Amortización**	**Uso**	**Total €**
HP ELITEONE G3 Intel Core i5-7500 (4 cores) a 4.50 GHz; 32 GB RAM	1000	3 años	6 meses	166,67
Monitor BenQ GW2480T 23.8"LED IPS FullHD	169	3 años	6 meses	28,17
Total				194,84

Tabla E.1: Costes del material

E.2 Costes de personal

En la Tabla E.2 se muestra el desglose de las horas dedicadas a las principales tareas realizadas durante el desarrollo del proyecto. El número total de horas es de 375 y su coste se detalla en la Tabla E.3.

Duración	**Hito**
60 horas	Estudio de las tecnologías, protocolos y dispositivos *software* involucrados en este proyecto.
65 horas	Estudio del código fuente del *software* switch BOFUSS.
110 horas	Implementación del modo de control in-band en el switch BOFUSS.
80 horas	Implementación de la reconfiguración del puerto de control del switch haciendo uso de los caminos generados por Amaru.
60 horas	Verificación de las funcionalidades implementadas en el switch BOFUSS

Tabla E.2: Desglose de las horas de trabajo

Concepto	Unidades	Precio/ud	Total €
Horas de ingeniería	375	30	11 250,00
Total			11 250,00

Tabla E.3: Coste de personal

E.3 Presupuesto total

El presupuesto final se calcula sumando los gastos de material, los gastos de personal y el Impuesto sobre el Valor Añadido (IVA) sobre la suma de los dos gastos. El presupuesto total se muestra en la Tabla E.4

Concepto	Total
Coste material *hardware*	194,84
Coste de personal	11 250,00
Subtotal	11 444,84
IVA (21 %)	2 403,42
Total	**13 848,26**

Tabla E.4: Presupuesto total

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá